System Architecture: the Context for Scenario-Based Model Synthesis

Sebastian Uchitel, Robert Chatley, Jeff Kramer, Jeff Magee
Department of Computing, Imperial College London
180 Queen's Gate, London, SW7 2BZ, UK
{su2, rbc, jk, jnm}@doc.ic.ac.uk

ABSTRACT

Constructing rigorous models for analysing the behaviour of concurrent and distributed systems is a complex task. Our aim is to facilitate model construction. Scenarios provide simple, intuitive, example based descriptions of the behaviour of component instances in the context of a simplified architecture instance. The specific architecture instance is generally chosen to provide sufficient context to indicate the expected behaviour of particular instances of component types to be used in the real system. Existing synthesis techniques provide mechanisms for building behaviour models for these simplified and specific architectural settings. However, the behaviour models required are those for the full generality of the system architecture, and not the simplified architecture used for scenarios. In this paper we exploit architectural information in the context of behaviour model synthesis from scenarios. Software architecture descriptions give the necessary contextual information so that component instance behaviour can be generalised to component type behaviour. Furthermore, architecture description languages can be used to describe the complex architectures in which the generalised behaviours need to be instantiated. Thus, architectural information used in conjunction with scenario-based model synthesis can support both model construction and elaboration, where the behaviour derived from simple architecture fragments can be instantiated in more complex ones.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications

General Terms

Design, Verification.

Keywords

MSCs, synthesis, architecture, generalisation

1. INTRODUCTION

Research has demonstrated that rigorous modelling of the behaviour of concurrent and distributed systems can prove to be very successful in uncovering design flaws. However, the impact that behaviour modelling has had among practitioners is limited. The construction of behaviour models remains a difficult task that requires expertise in formal systems such as process algebras and temporal logics. Our aim is to support the construction of formal behaviour models using synthesis techniques and languages that, while having rigorous semantics, are accessible to practitioners. In this paper we present an automated technique that builds behaviour models by combining the information from two complementary kinds of system specification: software architecture and scenarios.

SIGSOFT'04/FSE-12, Oct 31–Nov. 6, 2004, Newport Beach, CA, USA.

Software architecture describes the gross organisation of a system in terms of its components and their interactions [25]. Practitioners have found that architecture description languages (ADLs) can direct the construction of complex software systems and support reasoning about system design (e.g. [31]). To support rigorous behaviour analysis, ADLs have been extended or complemented with languages for describing component behaviour. For instance C2SADEL [23] uses logic while Wright [8] and PADL [5] use process algebra. Others, such as Darwin [18] and ACME [9], have been designed to be sufficiently abstract to support multiple views [15], one of which is the behavioural view (others are the service view for system construction and deployment, and the performance view for performance analysis). Darwin's behavioural view includes a process algebra: Finite State Processes (FSP) [19]

A limitation of existing approaches in supporting the behavioural view of software architectures is that the formalisms for describing component behaviour require considerable expertise in formal notations and semantics. In addition, these approaches require formal specification of intra-component behaviour, which practitioners generally have difficulty in producing. Many practitioners find that it is more intuitive to describe inter-component behaviour, i.e. component interactions at a system level (e.g. [7]). They also find it can be easier to describe behaviour in the context of a specific architectural instance rather than providing general context-less component type behaviour. Description by example can be a more manageable approach to the provision of initial descriptions of system behaviour rather than having to fully specify all components before building system models. In this paper we show that the behavioural view of software architectures can be constructed using scenario-based specifications and synthesis techniques.

Scenario-based specifications, such as message sequence charts (MSCs) [12] are popular as part of requirement specifications. Scenarios describe how system components, the environment and users interact in order to provide system level functionality. Their simple and intuitive graphical representation facilitates user involvement and makes them popular for conveying and

documenting requirements. The fact that scenarios are example-based specifications favours adoption by practitioners but introduces the problem of synthesising models that describes their composite behaviour. There have been a number of successful approaches to using scenario-based descriptions for building rigorous behaviour models [13, 14, 17, 27, 30, 32]. One of the main limitations of these approaches is that scenarios are essentially instance-level trace descriptions.

Scenarios are normally given at an instance-level: component instances interacting in specific instances of a system architecture. These architecture instances are often simplified fragments of the overall architecture of the system to be constructed. Simplified fragment architectures are chosen because they provide sufficient context to give satisfactory examples of how instances of the component types used in the real system are expected to behave. This is one of the strengths of scenario-based specifications. However, in terms of model synthesis, these instance-level descriptions introduce a serious challenge: the behaviour model that is required is not one of the simplified system being used to exemplify how component instances work. The aim is to build a behaviour model for a more complex system that is composed of some instances (among others) of the same component types as the simplified one, but in which the number of instances, and the way they are interconnected may be different.

What is needed is a synthesis technique that can *generalise* component type behaviour from instance-level examples in simple fragmentary architectural settings and can *instantiate* the component type behaviour into more complex architectural settings. In this paper we show how this can be achieved by exploiting architecture descriptions given in an ADL.

The organisation of the paper is as follows. Section 2 introduces scenarios and behaviour models. Using examples from an industrial case study section 3 discusses the limitations of using standard scenario-based model-synthesis. In sections 4 and 5 we introduce software architecture as a means for alleviating some of these limitations. Section 6 discusses the benefits of our approach together with related work. A tool implementing the approach described in the paper is available at http://www.doc.ic.ac.uk/ltsa/.

2. SCENARIOS AND BEHAVIOUR MODELS

The scenarios notation that we use is a syntactic subset of message sequence charts of the ITU standard [12]. We use basic message sequence charts (bMSC) to describe finite interactions between component instances (see Figure 1). Vertical lines represent component instances and horizontal arrows represent synchronous message exchanges between instances. We use high-level message sequence charts (HMSC) to describe alternative and iterative behaviour. An HMSC is a directed graph where nodes represent bMSCs and edges indicate their possible continuations. For the sake of brevity, we do not show hMSCs for the scenario specifications we discuss in this paper. Instead, all bMSCs are assumed to have an edge in the hMSC that goes from the bMSC to the initial node and an edge from the initial node to the bMSCs. In other words all scenarios in this paper denote iterative system behaviour, and if there are several scenarios, they are considered to be an initial choice within iterative system behaviour.

A formal semantics of the notation used is given in [28]. However, for the issues discussed in this paper an intuitive understanding of the semantics of bMSCs and hMSCs suffices. Briefly: we adopt a syntactic subset of the ITU standard [12] using trace semantics and weak sequential composition [12, 28]. BMSCs define a partial ordering of messages. The traces (sequences of message labels) specified by a bMSC are the total orderings of the bMSCs partial order. We also assume that messages model synchronous communication. It is important to note that the approach we present does not depend on these choices. In particular, it could be readily applied when asynchronous message communication is assumed.

The formalism used for modelling behaviour is that of labelled transitions systems (LTS). The graphical representation of LTSs is depicted in Figure 4. Parallel composition is used to model system behaviour resulting from composing LTSs that execute asynchronously but synchronize on all shared message labels. This notion of parallel composition is based on that of [11]. See [19] for a detailed explanation of LTS and parallel composition,

The textual language used to specify component behaviour is Finite State Processes (FSP) [19]. FSP is a process algebra whose semantics is given in terms of LTSs. The main FSP operators are action prefix "->", choice "|", parallel composition "||" and relabelling "/". Examples of FSP can be seen in Figure 5 and Figure 22.

3. SCENARIO-BASED MODEL SYNTHESIS

In this section we first introduce a simplified version of an industrial case study that serves as the basis for discussion throughout the paper. We then discuss current approaches to model synthesis, indicating their limitations.

3.1 Case Study

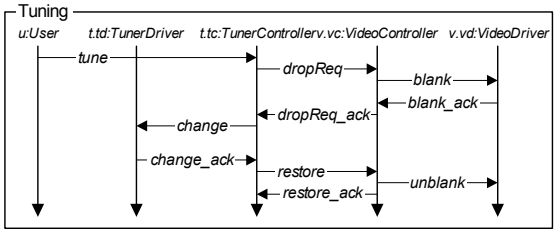


Figure 1 - Detailed Tuning scenario (1 tuner / 1 video output)

The case study involves modelling the behaviour of instances of a software architectural style used in a product family of Philips television sets. These TV sets can include multiple tuners and multiple video output devices that can be configured by the television user to display several signals in different configurations (picture-in-picture). The architectural style involves a horizontal communication protocol [31]. The protocol is concerned with controlling the signal path to avoid visual artefacts appearing on video outputs when a tuner is changing frequency. In Figure 1 we depict a simple scenario for a traditional TV set (1 tuner and 1 video output) that shows how controller components for the video and tuner devices coordinate in order to ensure that the video is not producing an output while the signal is being changed: when the tuner controller receives a command to change channel it requests (through a series of messages and acknowledgments) the screen to be blanked. Once blanked, the tuner controller requests the tuner driver to change

frequency. Finally, a series of interactions commands the video driver to restore its image; the image displayed will be on the new frequency being transmitted by the tuner.

The TV set used for Figure 1 is clearly not the target of the product family. The aim is to construct TVs with more than one tuner and more than one video output. For this, two additional components were developed: switch and fork. A switch component can interconnect two tuners with one video output, permitting the user to select which tuner is displayed on the output device. The fork permits a signal produced by a tuner to be split to two output devices. By combining forks and switches in a variety of architectures, different numbers of tuners can be connected in different ways to a set of output devices. Note that the names of the instances appearing in the scenario have been carefully chosen. The reason for this choice becomes apparent when we discuss model synthesis.

Figure 2 shows how a switch component works in the context of a 2-tuner 1-video output TV. Note that tuners and video components are described at a higher level of abstraction than in Figure 1: we do not show the subcomponents video controller and video device of component Video. Note also that we do not depict the hMSC that relates these scenarios temporally.

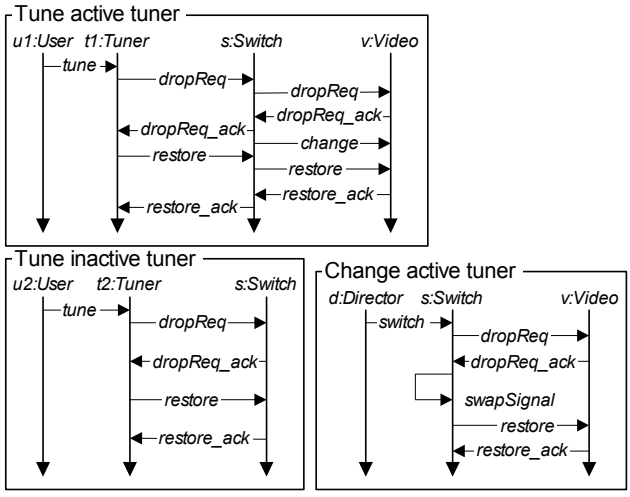


Figure 2 – Scenarios for Switch component

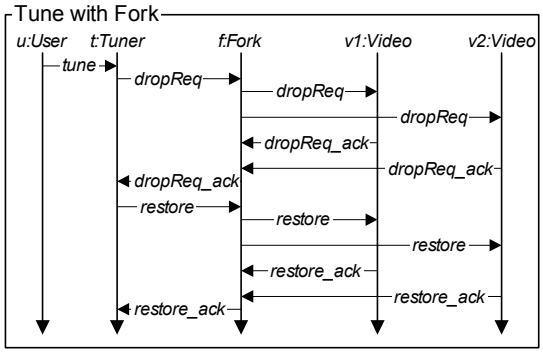


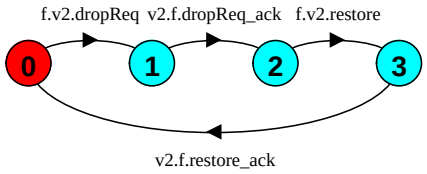
Figure 3 – Scenario for Fork component

The first scenario in Figure 2 shows what happens when a user changes the channel on the tuner that is currently controlling the signal that the video component receives. In this case, the switch component acts as an intermediary between the active tuner and the video forwarding messages of the horizontal protocol. The second scenario shows what happens when a user changes the channel on the tuner whose signal is not being displayed on the video output. In this case, the switch does not forward messages to the video component, but acts as the counterpart for the tuner, making it believe that there is a video component dropping its signal and then restoring it. The third scenario in Figure 2 shows the user commanding the switch to change the tuner whose signal is being displayed on video.

The behaviour of the Fork component is much simpler. In Figure 3 we show its behaviour in a setting in which there is one tuner and two video outputs: the fork simply forwards each message to both video components and waits for both of them to answer before it can respond back to the tuner.

3.2 Limitations of Scenario-based Synthesis

Some approaches to synthesis (such as [21, 32]) support the construction of component state-based models, but neglect the construction of a composite model that captures the behaviour of the components executing asynchronously and interacting through the messages or events described in the scenarios. That is, these approaches lack mechanisms for composing the component models into system level ones. Analysing system behaviour is crucial as it is in the interaction of components that many subtle composition issues arise. Approaches that use standard parallel composition notions require careful attention to the labelling or re-labelling of events in component models to guarantee that composition will correctly synchronise the component models [13, 16, 30]. For instance, from the scenario of Figure 3 our previous approach to synthesis [29, 30] generates models shown in Figures 4 and 5 where message labels have been prefixed with the sending and receiving instance names. Note how message dropReq has been relabelled to t.f.dropReq, f.v1.dropReq, f.v2.dropReq.




```
V2_Video = (f.v2.dropReq -> v2.f.dropReq_ack ->
   f.v2.restore -> v2.f.restore_ack -> V2_Video).
```

Figure 4 – FSP and LTS for the v2:Video component

For the sake of brevity, we do not give the details of the synthesis algorithms. Briefly, the algorithm builds an LTS that is the projection of the semantics of the scenario-based specification (which is a set of traces) onto the communicating alphabet of the component instance being synthesised.

This criteria for labelling transitions guarantees that when the LTSs are composed in parallel, the tuner synchronises with the fork on t.f.dropReq and that the video components do not do so. In fact, the synthesis approach in [29] guarantees that if the scenario-based specification is realisable [2], then the parallel composition of the instance components is trace equivalent to the traces determined by the semantics of the scenarios in Figure 3. The composition of the components for the system specified in Figure 3 is described in FSP as follows:

```
||SYSTEM = (V1_Video || V2_Video || T_Tuner
```

```
                || U_User || F_Fork).
```

Approaches to synthesis that work in the way described above (e.g. [13, 16, 30]) and also approaches to scenario semantics that use the same criteria ([12, 26] and [22]) produce a behaviour model for the system given by the scenario-based specification. The limitation is that the system being described in Figure 3 is not necessarily the system that is to be constructed. It is often the case that scenarios are given in simplified fragmentary settings; to ease explanation and reduce the number of scenarios needed to cover main behaviours. Typically, the system to be constructed is far more complex and includes more components of the same type as those that appear in the scenarios.

```
F_Fork = (t.f.dropReq -> f.v1.dropReq ->
   f.v2.dropReq -> v1.f.dropReq_ack ->
   v2.f.dropReq_ack -> f.t.dropReq_ack ->
   t.f.restore -> f.v1.restore -> f.v2.restore ->
   v1.f.restore_ack -> v2.f.restore_ack ->
   f.t.restore_ack -> F_Fork).

T_Tuner = (u.t.tune -> t.f.dropReq ->
   f.t.dropReq_ack -> t.f.restore ->
   f.t.restore_ack -> T_Tuner).
```

Figure 5– The t:Tuner and f:Fork synthesised from Figure 3

Suppose we want to build a television set with 2 tuners and 2 video outputs, that can be configured so that all combinations of tuners controlling video outputs are possible. Such a TV set would need to be implemented with 2 switches and 2 forks (see Figure 13, explained further in the next section). Suppose we wish to explore the behaviour of such a TV set, and see if the horizontal protocol implemented by the tuner, video controllers and drivers is adequate for such a context.

If we were to use a synthesis approach such as that described above, we would need to provide scenarios that explain how each of the forks, switches, tuner drivers, and tuner and video drivers and controllers behave and interact. The complexity of the scenarios needed would be unmanageable (See Figure 6 for a scenario for a simpler setting in which there is only one switch and one fork). In addition, the number of scenarios needed to cover a meaningful part of the system behaviour would certainly be unmanageable.

Clearly, for complex system architectures, the existing approaches to synthesis are impractical. What is needed is a synthesis technique that supports the use of scenarios that describe component instances in simpler settings and permits the construction of a behaviour model for a system in which component instances of the same type appear in more realistic, complex settings. For instance, how can we build a behaviour model for the 2-tuner 2-video output TV from the simpler scenarios given in Figures 1, 2 and 3? How can this be done considering that Figure 2 and 3 are at different levels of abstraction from Figure 1? How can we account for the fact that different instances appear in these scenarios (e.g., Figure 2 has tuners named t2 and t1, while Figure 3 has a tuner named t)?

The key notion here is system architecture. In the next sections, 4 and 5, we discuss our approach to architecture specification and explain how, by explicitly providing a simpler architecture for each scenario description, behaviour models for component types (rather than component instances) can be built. We show that, using an architecture description for the more complex target system, these component type behaviour models can then be instantiated to produce a model of the complex system (in our case the 2-tuner 2-video output TV).

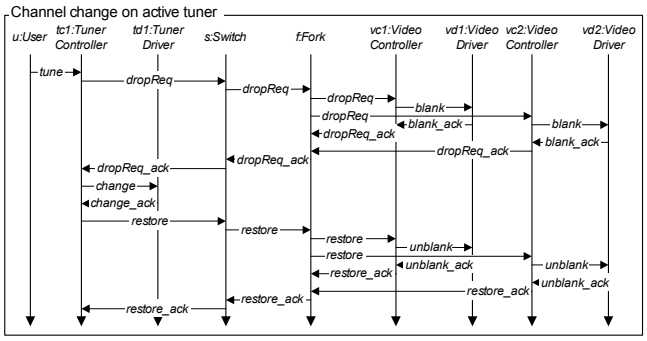


Figure 6 –2 tuner and video, 1 switch and fork TV scenario.

4. ARCHITECTURE

Darwin is a software architecture description language intended to bridge the gap between requirements and implementation in the design of complex systems. The emphasis in Darwin is to capture system structure. It has been designed to be sufficiently abstract to support multiple views, two of which are the behavioural view (for behaviour analysis) and the service view (for construction). Each view is an elaboration of the basic structural view: the skeleton upon which we hang the flesh of behaviour specification or service implementation [18]. Further views are those of performance and probabilistic behaviour. In the following we present an overview of Darwin.

4.1 Primitive Components

A primitive component is a component type with no component substructure. Component types (cf. classes) can be multiple instantiated (cf. objects). In the service view of architecture, a primitive component has an implementation defined by an object or objects programmed in a programming language. In the behavioural view, a primitive component is defined by a finite state labelled transition system.

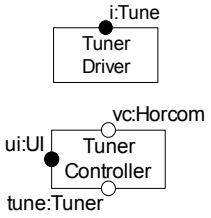
The example of Figure 7 depicts the Darwin graphical and textual description of two primitive component types with their interfaces. Tuner_Driver provides a service through its port `i` which is of interface type `Tune`. This means that the service will be provided by messages labelled `change` and `change_ack`. Tuner_Controller requires services through its ports `vc` and `tune` of interface types `Horcom` and `Tune` respectively, and provides a service through its port `ui` which is of interface type `UI`.

4.2 Composite Components

Darwin describes complex components in terms of the internal components that implement its services. A composite component is a component type with internal subcomponent instances and ports. Port bindings specify the interconnection between internal instances and between these instances and the composite component ports.

The composite Tuner component type (Figure 8) has one required and one provided service. A Tuner is composed from component instances of type Tuner Driver and Tuner Controller which are named `tc` and `td`. The ports `tune` and `i` of these

instances are bound together. In addition, ports `vc` and `ui` of the tuner controller instance are bound to the ports `horcom` and `ui` of the composite component. The fact that ports `tune` and `i` are not bound to a port of the composite component makes them inaccessible to any other component.

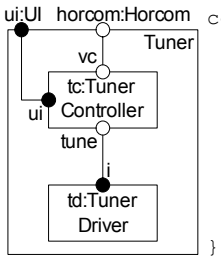



```
component Tuner_Driver {
   provide i:Tune;
}

component Tuner_Controller {
   provide ui:UI;
   require tune:Tune;
   require vc: Horcom;
}

interface UI {tune;}
interface Horcom {dropReq; dropReq_ack;
restore; restore_ack;}
interface Tune {change; change_ack; }
```

Figure 7 – Tuner Driver and Tuner Controller component types together with interface definitions




```
component Tuner {
    provide ui:UI;
    require horcom:Horcom;

    inst td:Tuner_Driver;
    inst tc:Tuner_Controller;

    bind tc.ui--ui;
    bind td.i--tc.tuner;
    bind tc.vc--hormcom;
}
```

Figure 8 – The composite component Tuner

A simple TV architecture `SimpleTV` (Figure 9) is a hierarchy, a composite component made out of instances of composite components `tuner` and `video` and a primitive component `User`. The `Video` component is defined in Figure 10 and uses primitive components `Video Controller` and `Video Driver`. For the sake of brevity, we omit the definitions of `Video` and `User`.

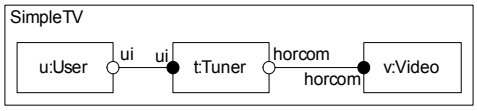


Figure 9 – A simple TV architecture

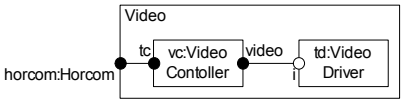


Figure 10 - Composite Component *Video*

Although not used here, Darwin also provides mechanisms for succinctly describing more complex systems by allowing parametric components and bindings and conditional and indexed instantiation. In addition, although in this example ports are bound to only one other port, in Darwin multiple required ports can be bound to a provided port; however, required ports may not be bound to more than one provided port.

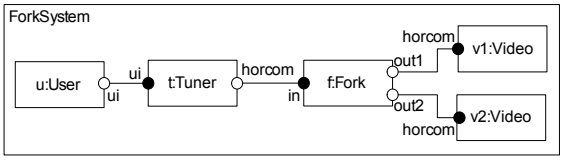


Figure 11 – Architecture for Simple Fork TV

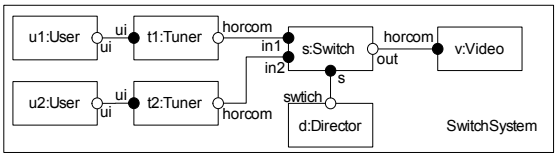


Figure 12 – Architecture for Simple Switch TV

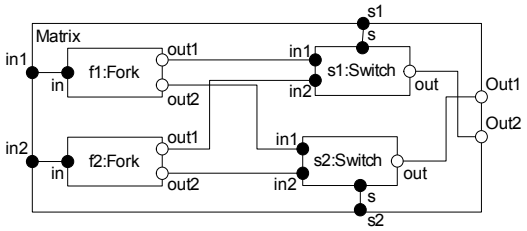


Figure 13 – Composite Component Matrix

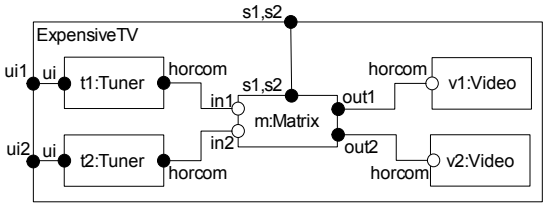


Figure 14 – Architecture for an Expensive TV

5. ARCHITECTURE IN MODEL SYNTHESIS

In this section we now show how we exploit architectural descriptions can be used in the context of scenario synthesis to *generalise* the instance-level behaviour of scenarios to produce component-type behaviour models. These models can then be *instantiated* in more complex architectures. In terms of our running example, we show how we can build a behaviour model for the 2-tuner 2-video output TV described in section 3. We use the scenario of Figure 1 in conjunction with the architecture of Figure 9, to generalise a behaviour model for component types `video controller`, `video driver`, `tuner controller`, and `tuner driver`. We use the scenarios in Figure 2 together with the architecture of Figure 12 to derive the behaviour of component type `switch`, and the scenario of Figure 3 with architecture of Figure 11 to construct the behaviour model for component type `fork`. We then instantiate the behaviour of these component types into the architecture of the 2-tuner 2-video output TV as described in Figure 13 and Figure 14.

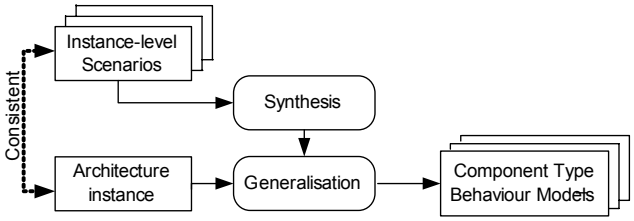


Figure 15 – The generalisation process.

5.1 Generalisation

Generalisation is a process (see Figure 15) that takes as input a set of instance level scenarios and an architecture instance that provides the structure of the system being described in the scenarios. The output of the generalisation process is a set of behaviour models corresponding to the component types of the instances that participate in the scenarios. There is a consistency requirement for generalisation that we discuss below.

The main issues when generalising instance level behaviour into component type behaviour are merging behaviours and translating messages directly between instances to interactions through ports. We deal with the latter first. For instance, if we are to synthesise a model for component type `video controller` from the scenario of Figure 1, then we need to discover through which port the `video controller` receives a message `drop_req`. From the definition of the `video controller` type it is simple to deduce that this must be port `tc` as it is the only one whose interface includes message `drop_req`. However, in general this reasoning may not suffice. It may be the case that a component can interact using the same message through several ports. This occurs even in the simplest of architectures: a pipe in a pipe and filter architecture interacts with its upstream and downstream filters through two different ports that have the same interface.

To understand through which port an interaction is taking place it is necessary to analyze which instances are involved in the interaction and to have a description of the overall structure of the instances participating in the scenario. Hence the problem to be solved is: given an architecture instance that describes the structure of the system whose behaviour is given by a scenario-based specification and given a message between two instances in the scenarios, a binding must be found within the architecture such that it binds ports of the two instances and that the interface of the binding includes the particular message label.

Consider Figure 9 as the architecture that describes the system of the scenario in Figure 1. In this architecture there is no binding between instances of components of types `Video Controller` and `Tuner Controller`. The problem is that the scenario in Figure 1 is being given at a different level of abstraction from the architecture of Figure 9. This is a common problem, scenario descriptions are typically not hierarchical, in that they may cut across several layers of abstraction, according to the detail in which the behaviour is being described; in contrast, architecture descriptions are hierarchical. To relate scenarios with architecture we must use the notion of a flattened (or elaborated) architecture configuration [20].

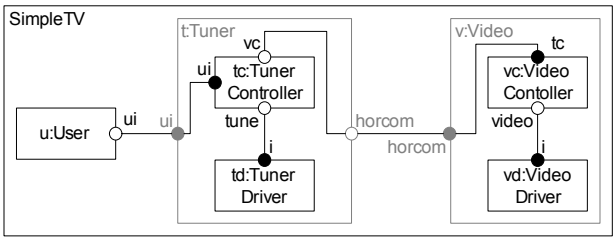


Figure 16 – Flat configuration of architecture in Figure 9.

A flat configuration is an architecture in which all composite components in its component hierarchy have been elaborated by their internal components and hierarchical bindings are replaced by direct bindings between primitive component instances. In other words, the hierarchy has been flattened. For example the architecture in Figure 16 is a flat configuration of the architecture in Figure 9. The elaborated composite components are shown in grey. Note that the instances in a flattened component instance are prefixed by the name of the enclosing composite. Consequently, instance `td` of component `Tuner Driver` that was part of instance `t` of `Tuner` becomes instance `t.td`.

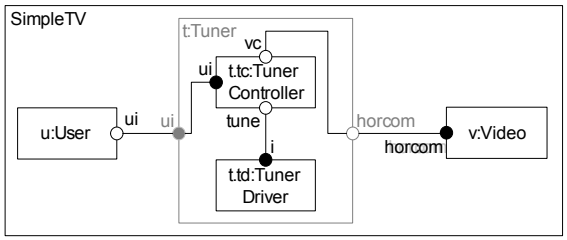


Figure 17 – A configuration of architecture in Figure 9.

A configuration is an architecture in which some of the composite components of its hierarchy have been flattened. Figure 17 is an example. Thus we have the following informal definitions:

Definition 1 (Elaboration Step) Replacing one composite component instance with its constituent component instances, renaming them and transforming the bindings to remove indirection.

Definition 2 (Configuration) A is a configuration of B if there are elaboration steps that can be applied to B to obtain A.

Definition 3 (Flat Configuration) A is a flat configuration of B if A is a configuration of B and no further elaboration steps can be applied to A.

Now consider Figure 9 as the architecture for Figure 1. The architecture does have a configuration in which we can find the instances that appear in the scenarios. The configuration Figure 16 not only has all the instances appearing in Figure 1 but also is consistent in the types of these instances. Furthermore, we can see that for each message in the scenarios, the instances interacting through the message have a binding in the configuration and that binding has an interface that includes the message label; for instance, the message `drop_req` between instances `t.tc` and `v.vc`. These instances appear in the configuration and have a binding between them. The binding connects ports `vc` and `tc` of interface `Horcom` which includes message label `drop_req` (see declaration of the `Horcom` interface in Figure 7). This holds for all messages in the scenario of Figure 1. Hence, we say that the architecture of Figure 9 and the scenario of Figure 1 are consistent.

Definition 4 (Consistency) An architecture instance D and a scenario specification S are consistent if there is a configuration D' of D such that

- If I is an instance of type T in S, then D' has an instance I of type T.
- If there is a message labelled *m* in S between two instances I and J, then D' has a binding between I and J whose interface includes *m*.

Given a scenario specification and an architecture that are consistent, it is simple to build a mapping from the messages sent

and received by one component instance to the ports through which these messages are sent. This mapping is the basis for generalisation. The mapping to ports for Figure 1 and architecture of Figure 9 is shown as port annotations in Figure 18.

Similarly, we can apply the same reasoning with the scenario of Figure 3 and the architecture of Figure 11 to build a mapping to component ports. Note that for this case we do not need to produce a configuration of the architecture as the level of abstraction of the scenarios and architecture coincide. If we then apply the mapping to the behaviour models of Figure 5 we obtain the models for component types given in Figure 19.

As we shall see in Section 5.4, we can instantiate the behaviour models for these component types into a complex architecture that has an arbitrary number of instances of these types and arbitrary binding structure. In particular we shall see how component type models can be instantiated for the architecture of Figure 14. However, we first discuss two issues that arise when generalising behaviour: model merging and ambiguity.

Note that achieving consistency between the scenario description and the architecture instance is a crucial task that must be done manually. For instance, the stakeholder providing the scenario of Figure 18 must make sure when providing a name for the tuner driver instance (`t.td:TunerDriver`) that the name is consistent with some configuration of the architecture instance of Figure 9.

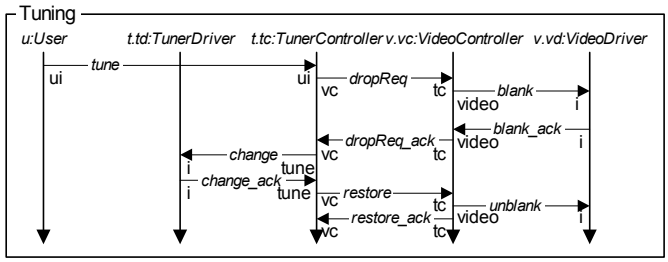


Figure 18 – Scenario annotated with inferred port names

```
Video = (horcom.dropReq -> horcom.dropReq_ack ->
   horcom.restore -> horcom.restore_ack -> Video).

Fork = (in.dropReq -> out1.dropReq -> out2.dropReq
   -> out1.dropReq_ack -> out2.dropReq_ack ->
   in.dropReq_ack -> in.restore -> out1.restore ->
   out2.restore -> out1.restore_ack ->
   out2.restore_ack -> in.restore_ack -> Fork).

Tuner = (ui.tune -> vc.dropReq -> vc.dropReq_ack
   -> vc.restore -> vc.restore_ack -> Tuner).
```

Figure 19 –Models for component types generalised from Figure 3 and Figure 11.

5.2 Model Merging

An important observation on the generalisation method described is that it builds one model for each instance participating in the scenarios. The difference with existing synthesis approaches is that these models have been generalised to port interactions. However, if we have more than one instance of the same component type in a scenario description, we build a generalised component model from each one and end up with multiple models for the same component type.

For instance from the scenarios in Figure 2 two behaviour models for component type `Tuner` would be synthesised, one from `t1:Tuner` and the other from `t2:Tuner`. In this particular case, the models would be equivalent, however this is not always the case. If different models for the same component type were generated, which one should be used? Because of the partial nature of scenario descriptions, the answer to this question is both. In fact, if we know from the behaviour of one instance that a component type can exhibit certain behaviour, and from another instance that the component type can exhibit some other behaviour, it is certain that the component type can exhibit both behaviours.

Hence, the final synthesis step that may be needed is to merge the multiple generalised models produced for a given component type. From our experimentation with this problem we have found that the naïve criteria for merging, which is producing a model that is a minimal trace refinement of both (i.e. can perform the union of the traces of both models to be merged) is too weak.

We are currently experimenting with a slightly stronger notion of merge that produces a model that can exhibit more traces than the minimal common trace refinement. The criterion is that of merging the initial states of all candidate models and then transforming the model to be deterministic while preserving traces. The reason we build a deterministic model is to be consistent with the semantics of HMSCs. Choices in HMSCs are defined as delayed choice [12], which essentially means that a component can delay the choice of which scenarios it is following while it is in the common prefix of the scenarios from which it has to choose. This is a reasonable assumption in the context of system behaviour elicitation where the interest lies in the traces that a system can perform and not in the internal branching structure of the components. Note that it is straightforward to prove that merging initial states above preserves simulation [11] and consequently trace inclusion.

5.3 Resolving Ambiguity

The second issue that arises when generalising instance level behaviour to component type models is that sometimes the information available is not strong enough to infer a unique translation from a message to a port. Suppose we have an architecture A and a scenario specification S that are consistent. Suppose that A' is the configuration for which all instances in S can be found and for which all messages between instances have their corresponding binding. If there is a message labelled m in S between two instances I and J for which A' has more than one binding between I and J and whose interface includes m, then we have an ambiguity and need to select between ports.

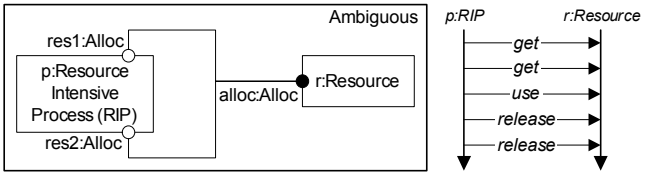


Figure 20 - Example of ambiguity.

As an example consider Figure 20, the scenario and architecture where interface `Alloc` is defined as `{get; use; release}`. The scenario depicts a component `p` that requires two resources `res1` and `res2`. However, rather than binding these two

required ports to different resource providers, they are bound to a component r that is capable of providing at least the two resources that p requires. Deriving ports for messages sent by component p is not possible. For each message there are two ports through which p could be interacting: res1 and res2. This is not the case for r, where it is clear that it always interacts through port alloc. Messages in the scenario of Figure 20. In these cases validation with the user is necessary to allow generalising the model for component type RIP.

Once a user resolves the ambiguity, the information should be recorded explicitly. One way could be to annotate the scenarios with the port names. This extension to sequence chart notations is consistent with that of other authors, such as [6]. Currently, we have not extended our approach to support annotated scenarios with ports. We wish to experiment further to assess whether these situations arise in practice (up to now, we have found that ambiguities are rare, in our running example of television sets and other case studies we there are none) and to investigate other solutions to this problem.

5.4 Instantiation

The previous subsection discusses generalisation of component instance behaviour to component type behaviour. The goal here is to build a model for a complex architecture from the models of components types. This process involves instantiating the models for component types into models for the specific component instances that are part of the target architecture, and relabelling actions of the instantiated models to guarantee that instances interact correctly.

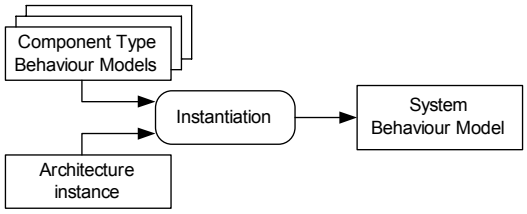


Figure 21 – Inputs and outputs of the instantiation process

The instantiation process is performed top down through the component hierarchy of the architecture. Each composite component is constructed by instantiating its composing instances and performing the appropriate relabellings [20].

For the complex TV of Figure 14, we must instantiate a matrix component m, two tuners t1 and t2, and two video components v1 and v2. FSP provides operators for instantiating instances of a model. Given a process P, a:P is a process that prefixes all labels in P with a. Hence X:P where X is a set of labels, produces a process x:P for each x in X. Hence if we are to build the parallel composition of the instances in Figure 14 the FSP is:

```
({t1,t2}:Tuner||{v1,v2}:Video||m:Matrix).
```

In order to achieve communication between the instances of the Expensive TV, messages must have the same labels. For example according to Figure 14, the Matrix component m can interact through its in1 port on action drop_req with the tuner component t1 through its horcom port. Hence, we must ensure that messages labelled m.in1.drop_req are relabelled to t1.horcom.drop_req. Moreover, all messages prefixed m.in1 should have their prefix relabelled to t1.horcom .

FSP provides an operator that performs prefix relabelling. If P is a process, X a set of labels and y a label, then P/{X/y} produces a process in which each transition with a label that is prefixed with y is replaced by a set of transitions where y has been replaced by each element in X. Hence, the instantiation of Figure 14 is:

Note that because a provided port may be bound to several required ports, the relabelling must always be performed from the provided port into the set of required ports that are bound to it. Relabelling in the other direction would in general not necessarily produce correct instantiations (see [20] for a full discussion).

In principle, instantiation process continues recursively down the component hierarchy until all composite components have been instantiated in terms of their internal instances and these are primitive components. In the example above, the instantiation process would continue on Matrix, Video and Tuner and stop when primitive component fork, switch, and tuner and video controller and driver are reached

However, in the context of scenario synthesis, it may be the case, that no scenarios have been provided beyond a certain level of detail. For instance, assume that the scenario of Figure 1 is not available. In this case, when instantiating the ExpensiveTV we would need to use the component type model synthesised for Tuner and Video from the scenarios in Figure 2 and Figure 3 (probably merging them before hand). Hence, the instantiation process would stop before the hierarchy of the ExpensiveTV is entirely flattened.

```
||ExpensiveTV = ({t1,t2}:Tuner ||
                {v1,v2}:Video || m:Matrix)
     /{t1.horcom/ m.in1, t2.horcom/m.in2,
     m.out1/v1.horcom, m.out2/v2.horcom}.
```

Figure 22 - FSP for the ExpensiveTV

6. DISCUSSION AND RELATED WORK

The behaviour model synthesised by generalising five scenarios in three architectural contexts (Figure 1 in Figure 9, Figure 2 in Figure 12 and Figure 3 in Figure 11) and then instantiating the Expensive TV of Figure 14 results in a state space of over 70000. This shows the potential that exploiting architecture descriptions has in scenario synthesis. The number of scenarios needed to construct a portion of such model using traditional instance-level synthesis techniques would certainly be significantly larger and unmanageable.

The approach presented in this paper is in line with the way in which scenarios are used: different simplified, possibly fragmentary, architectural settings are chosen to describe the behaviour of different component types. The instances of a scenario description are usually not intended to be actual instances of the more complex target system; they are intended to provide concrete settings in which to exemplify a specific aspect of component behaviour [7]. The synthesis approach we present can handle these different architectural settings at different levels of abstraction elegantly by exploiting architectural descriptions.

The case study to which we applied our synthesis technique is a simplified version of the horizontal communications protocol for a Phillips TV product family. In reality, tuner and video controllers support a more decoupled protocol which allows drivers to signal controllers once they have completed their tasks.

By specifying the complete protocol in a simple context, and by instantiating complex TV sets, we were able to incrementally elaborate the scenarios and describe the intended behaviour of the switch component, the most complex of all the primitive components. We discovered that our switch component could not handle appropriately the more decoupled protocol in the context of the `ExpensiveTV`. The model for the `ExpensiveTV` resulted in deadlocks that indicated where the switch was not handling certain states of the two tuners and video components. In fact, ten additional scenarios were required to achieve the complete behaviour of the switch component type. Note that these scenarios had not arisen in the simple setting in which we described the switch. This is an example of how generalisation and instantiation can prompt further behaviour elicitation.

Note that the benefits obtained by exploiting architecture in synthesis are not for free. As explained in Section 5.1, careful attention must be given when providing scenarios with their architectural context. The naming of instances in scenarios must coincide with some configuration of the architectural instance associated with the scenarios. This task must currently be done manually.

The ideas we present in this paper are orthogonal to the assumptions and scenario notations taken by existing synthesis approaches such as [10, 16, 21, 32]. We believe they could be extended to incorporate the ideas presented here. We do not mandate how the behaviour models are constructed from the scenarios. However, we do indicate how the inference of labels should be made in order to achieve models for component types rather than for component instances and how these component type models can be instantiated into complex architectures.

ADLs other than Darwin, such as ACME [9], could be used in the approach, though the translation from ADL structure to model composition expressions may not be as straightforward. Darwin and FSP were designed to be complementary which facilitates this translation.

The generalisation-instantiation process we present does not change the semantics of synthesis if two conditions hold: first, scenarios are instantiated into the same architecture from which they were generalised, and second there is not more than one component instance per type in the scenarios. In the context of the synthesis approach in [30] we can show that applying generalisation and instantiation to a scenario specification S with a consistent architecture A in which there are no two instances of the same type in S, will produce the same as not performing the generalisation/instantiation altogether.

If there are multiple instances of the same type, then generalising and instantiating the scenarios from and to the same architecture can result in additional traces that correspond to scenarios that are symmetric to existing ones. In other words, scenarios in which one instance of a component type displays behaviour that was originally specified for a different instance of the same type in the original scenarios. This is caused by the generalisation process merging behaviours of instances of the same type. Detecting these symmetries may prompt further elaboration of scenarios and architecture descriptions.

There are many approaches to scenario synthesis. However, to the best of our knowledge none of them use architectural information in order to synthesise the behaviour of component types rather than that of instances. Neither can existing approaches combine behavioural information given in different scenarios at different levels of abstraction to build system behaviour models. In [32], Whittle and Schumman present an approach to synthesis and show how a statechart model for an ATM can be built. The authors do not show how to instantiate the model in a setting where multiple ATMs interact with the same bank. Other approaches to synthesis such as in [16, 21] do not use any structural or architectural information either. SDL architecture descriptions are used in the context of synthesis of SDL from MSCs in [1], however, the information is not exploited to build component type behaviour. Synthesis produces SDL models at an instance level. Hence, scenarios must be given in the true architecture of the system rather than in simplified settings. In the Fujaba approach [10], structural information is combined with activity diagrams to synthesise class behaviour. However, the focus is not on constructing system behaviour models for software architecture analysis but on code synthesis and object orientation. In [17] some architectural information is extracted from scenarios to improve synthesis, however, there is no generalisation and subsequent instantiation of the behaviour in scenarios into different architectural settings.

The approach presented in this paper could potentially be applied to collaboration diagrams [24]. Collaboration diagrams can be used to synthesis state-based models much in the same way as scenario notations. These diagrams focus on system structure and graphically overlay on the structure sequences of interactions. As with scenarios, to be able to use the information in these diagrams to construct behaviour models of complex architectures, behaviour must first be generalised for classes.

Techniques for the analysis of scenario-based specifications (e.g. [2-4]) do not use architectural information either. Future work should address how architectural information could be used in this context.

Merging of behaviour models has not been extensively studied in the process algebra community. However, it is essential when reasoning about behaviour in the early requirements phase. Models from different viewpoints may need to be merged, or as in this paper, models from different instances of the same component type. The criteria we use for merging models (merging their initial state) coincides with the criteria used by Kruger et al. [16] and others ([14, 27]). However, because of the more expressive scenario notations used in these approaches, typically more states are merged. An important difference with Kruger et al. [16] is that they do not merge complex behaviour models; their models correspond to total orders and they use state information to link the first and last states of these total orders together. A fundamental difference is that merging in [16] and [27] is not used to put together multiple descriptions of the same component type but rather as an artefact to cope with state labels in scenario descriptions. We are currently investigating model merging in the context of modal transition systems [33]; we expect to apply this work in the context of synthesis in the future.

7. CONCLUSION

We have presented an approach that exploits architectural information in the context of behaviour model synthesis from scenarios. We have shown that exploiting architecture descriptions can support construction of complex behaviour models. Our approach does not require behaviour to be described

in the context of the system to be built. Rather, it allows behaviour to be described in scenarios that use simplified architectures (possibly at different levels of abstraction and including different numbers and types of instances) selected specifically to facilitate the description of particular aspects of component behaviour.

We have exemplified our approach with a simplified version of an industrial case study. The use of the more complex version of the case study to validate our approach has shown that using architecture can support the construction of complex of models using scenario synthesis. It has also shown that the complexity of the scenarios needed to produce such models is significantly lower, because behaviour can be described in simpler architectural settings. More experience is needed to confirm this.

Finally, this work supports the hypothesis that requirements and architectures should be developed hand in hand. In this paper we do not discuss the tool we have developed to support our approach. The tool is available at http://www.doc.ic.ac.uk/ltsa/.

8. ACKNOWLEDGMENTS

The authors acknowledge that this research was supported in part by the STATUS ESPIRIT project (IST-2001-32298) and EPSRC grant READS GR/S03270/01.

9. REFERENCES

1. Abdalla, M.M., Khendek, F. and Butler, G., New Results on Deriving SDL Specifications from MSCs in *SDL Forum '99*, (1999).
2. Alur, R., Etessami, K. and Yannakakis, M., Inference of Message Sequence Charts in *22nd IEEE International Conference on Software Engineering*, 2000, 304-313.
3. Alur, R., Holzmann, G.J. and Peled, D., An Analyser for Message Sequence Charts in *2nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, (Passau, 1996), Springer, 35-48.
4. Ben-Abdhallah, H. and Leue, S. Syntactic Analysis of Message Sequence Chart Specifications, Electrical and Computer Engineering, University of Waterloo, 1996.
5. Bernardo, M., Ciancarini, P. and Donatiello, L. Architecting families of software systems with process algebras. *ACM TOSEM*, *11* (4). 386-426.
6. Bordeleau, F. A Systematic and Traceable Progression from Scenario Models to Communicating Hierarchical Finite State Machines *SCS*, Carleton University, Ottawa, 1999.
7. Carrol, J.M. (ed.), *Scenario-based design: envisioning work and technology in system development*. Wiley, New York, 1995.
8. Garlan, D., Allen, R. and Ockerbloom, J., Exploiting Style in Architectural Design Environments in *Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, (New Orleans, 1994), 175-188.
9. Garlan, D., Monroe, R. and Wile, D., ACME: An Architecture Description Interchange Language in *CASCON'97*, (1997).
10. Giese, H., Tichy, M., Burmester, S., Schäfer, W. and Flake, S., Towards the Compositional Verification of Real-Time UML Designs in *ACM ESEC/FSE*, (Helsinki, 2003), ACM.
11. Hoare, C.A.R. *Communicating Sequential Processes*. Prentice Hall, Englewood Cliffs, New Jersey, 1985.
12. ITU. Message Sequence Charts, International Telecommunications Union. Telecommunication Standardisation Sector, 1996.
13. Khriss, I., Elkoutbi, M. and Keller, R., Automating the Synthesis of UML StateChart Diagrams from Multiple Collaboration Diagrams in *UML'98: Beyond the Notation*, (1999), Springer-Verlag, 132-147.
14. Koskimies, K., Männistö, T., Systä, T. and Tuonmi, J. Automated Support for Modeling OO Software. *IEEE Software*, *15* (1). 87-94.
15. Kruchten, P. The 4+1 View Model of Architecture. *IEEE Software*, *12* (6). 42-50.
16. Krüger, I., Grosu, R., Scholz, P. and Broy, M. From MSCs to Statecharts. in Rammig, F.J. ed. *Distributed and Parallel Embedded Systems*, Kluwer, 1999, 61-71.
17. Leue, S., Mehrmann, L. and Rezai, M., Synthesizing ROOM Models from Message Sequence Charts Specifications in *13th IEEE Conference on Automated Software Engineering (ASE'98)*, (Honolulu, 1998), IEEE CS, 192-195.
18. Magee, J., Dulay, N., Eisenbach, S. and Kramer, J., Specifying Distributed Software Architecture in *ESEC'95*, (1995).
19. Magee, J. and Kramer, J. *Concurrency: State Models and Java Programs*. John Wiley & Sons Ltd., New York, 1999.
20. Magee, J., Kramer, J. and Giannakopoulou, D., Analysing the Behaviour of Distributed Software Architectures: a Case Study in *5th IEEE Workshop on Future Trends of Distributed Computing Systems*, (Tunis, 1997), 240-245.
21. Mäkinen, E. and Systä, T., MAS – An Interactive Synthesizer to Support Behavioral Modeling in UML, in *23rd IEEE International Conference on Software Engineering (ICSE '01)*, (Toronto, 2001), 15-24.
22. Mauw, S. and Reniers, M.A., Refinement in Interworkings in *CONCUR'96*, (Psia, 1996), Springer, 671-686.
23. Medvidovic, N., Rosenblum, D.S. and Taylor, R.N., A Language and Environment for Architecture-Based Software Development and Evolution in *21st International Conference Software Engineering (ICSE '99)*, (1999), 44-53.
24. OMG. The Unified Modeling Language 2.0, 2003.
25. Perry, D.E. and Wolf, A.L. Foundations for the Study of Software Architectures. *ACM SIGSOFT Software Engineering Notes*, *17* (4). 40-52.
26. Reniers, M.A. Message Sequence Chart. Syntax and Semantics, Eindhoven University of Technology, Eindhoven, 1999.
27. Somé, S., Dssouli, R. and Vaucher, J., From Scenarios to Timed Automata: Building Specifications from User Requirements in *Asia Pacific Software Engineering Conference (APSEC'95)*, (1995), 48-57.
28. Uchitel, S. Elaboration of Behaviour Models and Scenario Based Specifications using Implied Scenarios *Department of Computing*, Imperial College London, 2003.
29. Uchitel, S., Kramer, J. and Magee, J. Incremental Elaboration of Scenario-Based Specifications and Behaviour Models using Implied Scenarios. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *To appear*.
30. Uchitel, S., Kramer, J. and Magee, J. Synthesis of Behavioural Models from Scenarios. *IEEE Transactions on Software Engineering*, *29* (2). 99-115.
31. van Ommering, R., van der Linden, F., Kramer, J. and Magee, J. The Koala Component Model for Consumer Electronics Software. *IEEE Computer*, *33* (3). 78-85.