Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

ELSEVIER Electronic Notes in Theoretical Computer Science 108 (2004) 83–98

www.elsevier.com/locate/entcs

Generating Snapshots of a Component Setting

Sotiris Moschoyiannis[1]

Department of Computing
University of Surrey
Guildford, Surrey
GU2 7XH, England

Abstract

Software components are often seen as panacea when faced with the challenges of the increasing use of software in many diverse areas of computation. However, the complex 'call interplay' at the interfaces between components often results in pathological behaviour and hinders effective reuse. There is a clear need for languages for documenting and specifying components in such a way that insulates them from changes in the configuration. In this paper we describe the use of Live Sequence Charts (LSCs) to describe component interactions. Then, we advocate an approach for formalising those interactions to ensure that components interact properly whilst making minimal assumptions about their neighbours.

Keywords: interactions, behaviour, LSCs, vector semantics

1 Introduction

Component-based software development has received considerable attention over the recent years as a means for dealing with the ever changing requirements of modern software systems. Especially in the context of embedded software, new products require increasingly complex software to be developed in a timely and affordable fashion. Further, the configuration is likely to change not only for new products but even for a single product during development [14]. Modern software systems often comprise complex combinations of existing functionality. As a result, there is a need to reuse software that

[1] Email: s.moschoyiannis@eim.surrey.ac.uk

doi:10.1016/j.entcs.2004.01.014

has been developed by different developers, under differing assumptions about the behaviours between components where inconsistencies tend to occur. This poses further challenges to the reuse of software components and underlines the need for this activity to be organised systematically.

The prevalent paradigm towards this end is that of component-based design. Systematic reuse of components requires both interface specification of components and methods for conducting compatibility checks for component interfaces. In this paper, we are concerned with the former. We describe the use of Live Sequence Charts (LSCs) [2] to specify the allowed interactions between components in a scenario-based fashion. Then, we present a formalism for capturing component interactions, with regard to specific scenarios.

We are interested in modelling the occurrences of events at component interfaces: the order in which the services provided by a component are requested as well as the order in which the component requests services from other components. Our modelling approach aids formal analysis and reasoning about component interactions. We also hint towards extracting state-based information from the message interplay between components. This is particularly useful when considering rapidly changing configurations to accommodate new products or systems.

This paper, is organised as follows. In Section 2 we present an example taken from embedded software for analogue televisions and use universal LSCs to capture specific scenarios with regard to the particular configuration of components. The example is extended in Section 3 where the configuration is changed to incorporate a new feature. In Section 4 we present a formalism for interacting components that supports change. The paper finishes with some concluding remarks and ideas for future work.

2 Modelling horizontal communication

Typically, Message Sequence Charts (MSCs) [16] are used to specify scenarios as sequences of message interactions between objects or processes. However, MSCs often do not tell the complete story. Their interpretation can be ambiguous; for instance, does an MSC describe all behaviours or a set of sample behaviours of a system? According to the ITU standard [16] MSCs only do the latter. Then, virtually nothing can be said in MSCs about what the system will do when the described scenario actually occurs. LSCs [2] resolve such issues as they explicitly distinguish between *mandatory* and *possible* behaviour. This is done by adding liveness to individual parts of the charts.

Within a chart, the live elements, termed *hot* describe mandatory behaviour - things that *must* occur. When used properly, hot elements can

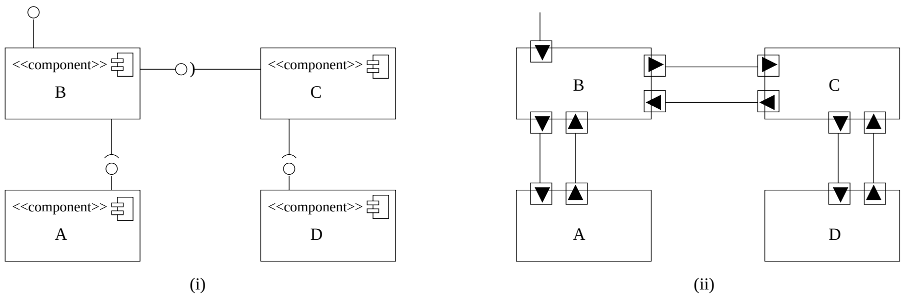

Fig. 1. Component spec architecture using UML 2.0 in (i) and Koala in (ii)

describe forbidden behaviour, i.e. disallowed sequences of interactions. Other elements, termed *cold*, describe possible behaviour - things that *may* occur. They can be used to capture conditional behaviour and various forms of iteration. In terms of notation, all hot elements are indicated in solid lines/boxes/hexagons while cold elements in dashed lines/boxes/hexagons. We return to the notation of LSC constructs in discussing our example below.

In what follows we present an example taken from the domain of analogue televisions. The example is essentially a small version of the one in [14] but contains the necessary detail to help us illustrate our approach towards capturing interactions between components.

In [14] the authors describe problems that arise when composing control software in the context of embedded systems used in consumer electronics (CE) products. Many control tasks in a TV, for instance, coordinate devices in the same signal path. This implies a strong dependency upon the topology of the hardware, which is subject to change in new products but also for the same product during its development. In light of such problems the approach taken in [14] is to allow components to communicate using horizontal communication interfaces in addition to the vertical control interfaces. The idea is that components controlling individual hardware devices have input and output ports that mirror the hardware and communicate through those.

In a television set a common task is that of *tuning*. When the frequency of the tuner changes it produces noise which might result in undesired artifacts on the TV screen. Therefore the screen of the television should be blanked before the frequency is changed. As soon as the tuner is tuned to the new frequency, the screen can be unblanked (see also [14]). Figure 1 shows the configuration of components in a TV platform required for this functionality, where component A is the tuner driver, B is the 'intelligent' component controlling the tuner, C is the 'intelligent' component controlling the video output, and D is the video output driver.

Figure 1 (ii) shows the configuration using the notation of the Koala component model [15]. Koala's graphical notation strongly resembles hardware design. Software components are rendered as IC chips and interfaces are represented as pins of the chip. The idea is that components have input and output ports through which components send and receive signals. Triangles on connection points designate the direction of function calls.

Subfigure (i) of Figure 1 shows the same configuration this time using UML, and in particular considering the introduction of component or composite structure diagrams in UML 2.0 [10]. Compared to UML 1.x, there is strong indication in the draft adopted specification of UML 2.0 of a shift of focus from implementation to specification with regard to components.

2.1 Modelling mandatory behaviour

The task we are concerned with here is that of tuning. Suppose that the top-level control software (not included in Figure 1) wants to change the frequency. It issues a request *tune(f)* on the control interface of B. In order to provide that service, B requires C to drop its signal. C in turn, requires the blanking of D before it can drop its signal. Once C has indeed dropped its signal, B can proceed to request the tuner driver component A to change its frequency. Once A fulfills that request, B issues a new request on C to restore its signal. Again, C requires D to unblank before it can complete the restore request of B. Once C is restored, B may acknowledge the change frequency request to the control software component. This completes the call to *tune(f)*.

It can be seen that a number of interactions take place whenever the tuner changes frequency. We illustrate these interactions in more detail using the notation of LSCs [2]. To model the interaction of our example, we use a universal LSC as we want to describe mandatory behaviour; what *must* happen. It induces an action-reaction relationship between its prechart (annotated with a dashed borderline hexagon; it is essentially a cold condition) and its main body (annotated with a solid borderline rectangle) as shown in Figure 2.

Here, we assume a reliable communication medium, in the sense that no messages can be lost. This view is consistent with the particular application domain, that of embedded software for CE products. Further, all messages are considered synchronous in the sense that no significant time elapses between sending and receiving a message (emitting and absorbing a signal). Closed triangular arrowheads are used to denote synchronous messages, see Figure 2.

The chart, shown in Figure 2, starts with a prechart where the control software issues a request to change frequency on B; that is, message $tune(f)$ on the LSC. Note that we use an actor to represent the top-level control software. We want to stress the fact that the top-level control software is the external entity / instance that initiates the interaction that follows, but does

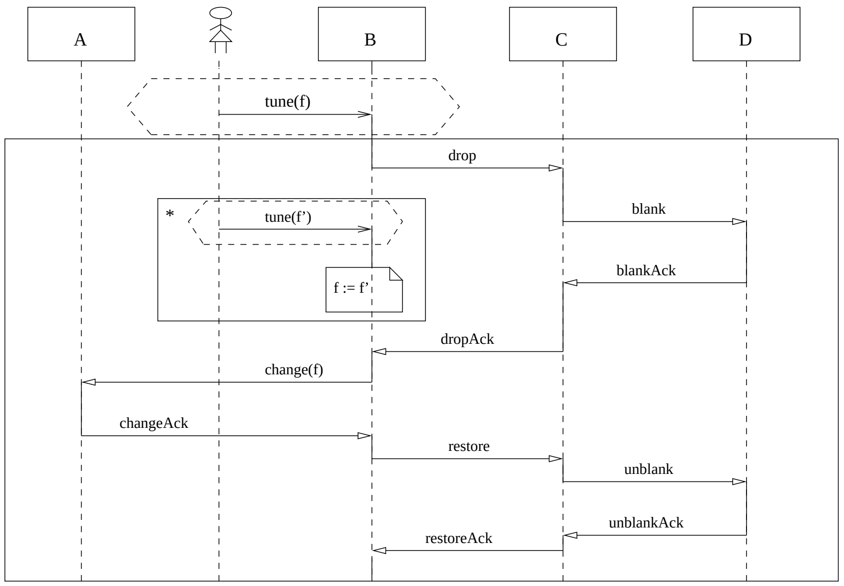

Fig. 2. An LSC for changing the frequency of the tuner

not participate in it as such.

After the control software issues a tuning request, the main body of the chart is entered. This is indicated by the solid part of the lifeline of B following the receiving of a tuning request. Components A, C and D must also proceed to the body of the chart. The main body starts with B's reaction to what happened in the prechart. B requests C to drop its signal (message *drop* on the LSC). Before C can drop its signal though, it must coordinate its downstream device, i.e. the video output driver component D. In other words, it must request the blanking of D. It does so by sending *blank*. After some time, D sends a blanking acknowledgement to C to indicate successful completion of the blanking operation. Only then can C acknowledge B's drop request.

Notice that the messages *drop*, *blank*, *blankAck* are synchronous while $tune(f)$ is asynchronous, as indicated by its open ended arrowhead. This means that the actor can engage in other tasks while the tuning operation is carried out. That includes issuing a new $tune(f')$, $f' \neq f$, to component B while the tuning operation is still in progress. In fact, the actor or top-level control software could issue a number of new tuning requests. If a new $tune(f')$ indeed occurs during that period (before completion of the drop request), then B must remember the new frequency value for future use in $change(f)$. A $tune(f')$ can only occur before C acknowledges the drop request of B, and thus concurrently with the blanking operation going on between C and D.

This situation is modelled using an unbounded loop (denoted by a subchart with a '*' on the top left corner). An unbounded loop iterates forever and can only be exited if a cold condition inside it evaluates to false. In general, if the cold condition is placed on the very begining of the loop subchart, then we are modelling a *while* loop. If the cold condition is placed at the bottom of the loop subchart, then we are modelling a *do-until* loop. We do the former in our example. In the LSC of Figure 2 we use a cold condition (denoted by a dashed hexagon) which is precisely the occurrence of a $tune(f')$, $f' \neq f$. If this message does occur, then the cold condition evaluates to true and the loop subchart is entered. Inside the loop, B updates the frequency value (the assignment operation is annotated with a rectangle with a flipped edge, in a fashion similar to the note construct in UML). If a new $tune(f')$ does not occur, the loop is ignored and execution of the scenario continues with what follows immediately underneath the loop.

Notice that the placement of the loop subchart is also important. A new $tune(f')$ can only occur after the $drop$ message and before the $dropAck$ message. This is imposed by the partial order induced along the lifeline of B. The number of iterations is determined by the number of occurrences of new tuning requests, which take place while the drop operation - and its 'nested' blanking operation - is still in progress. In more simple terms, a new $tune(f')$ cannot occur after $dropAck$ has been received by B. A new tuning request after that point is considered a disallowed sequence of events in the scenario and essentially corresponds to *forbidden* behaviour. We further discuss this situation in the following subsection.

The activity described in the loop can be going on concurrently with the blanking operation between C and D. We do not need to denote this concurrency explicitly on the LSC because the loop and the blanking operation involve different components. If some component was participating in both, we could use the coregion notation (dotted vertical lines around the concurrent messages) with the true-concurrent interpretation of [5]. We return to a discussion on coregions in Section 3.

Now returning to the example, once B receives the drop acknowledgement ($dropAck$) from C, it proceeds (it *must* proceed as its lifeline is solid) to send a change frequency message to A. After a while, A indicates completion of the change request by sending $changeAck$. B can then issue a restore request to C. Again C has to coordinate its downstream device, so it requests the unblanking of D. After again some time (of the magnitude of msec), D acknowledges the unblanking request. C then acknowledges the restore request of B and this completes the change frequency request of the control software. The TV is now tuned to the new frequency.

Notice that acknowledgements in our example are of significant impor-

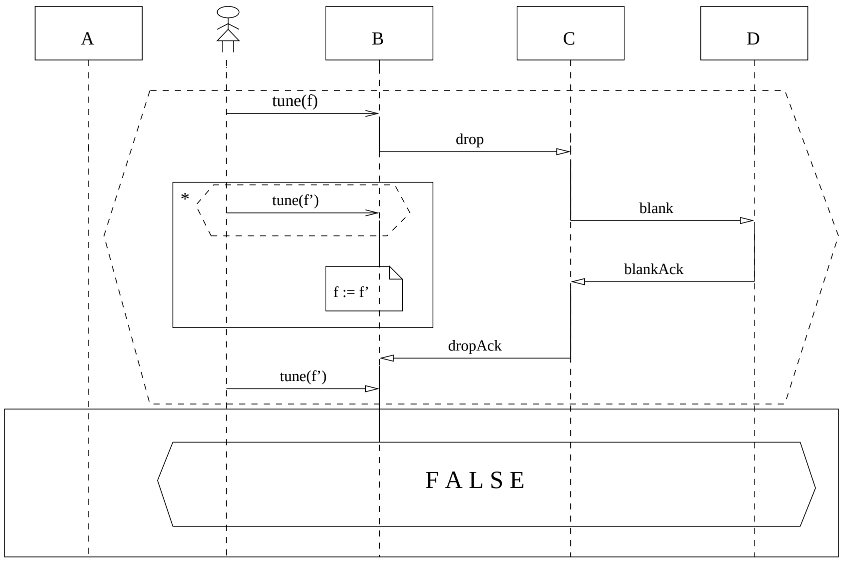

Fig. 3. New tune requests cannot occur after the drop operation has been completed

tance. In contrast to typical acknowledgements they do not merely acknowledge the receipt of a message, but rather indicate that the task or functionality requested has been completed. We are modelling the 'delayed approval' case here but our explanations apply equally well to the 'immediate approval' case too. In the domain of analogue televisions an acknowledgement is actually an up call to an upstream device to indicate that the requested operation has indeed been completed. Upstream devices are those closer to the source of the signal and downstream devices those further away from it (see [14,15]).

2.2 *Modelling forbidden behaviour*

Up to this point we have described the use of LSCs for modelling mandatory behaviour in a component setting. With LSCs it is also possible to describe forbidden behaviour. That is, behaviour that the system is not allowed or at least not intended to exhibit within a given scenario. When we want to forbid a certain sequence of communications, we specify these communications in the prechart of an LSC and place a hot false condition in the main body of the chart. Recall that if a hot condition evaluates to false then the whole chart is aborted and an error is flagged. The reserved word *FALSE* in LSCs [4] can be used for this purpose as it makes the condition it is placed in always evaluate to false.

The LSC shown in Figure 3 says that a new $tune(f')$ cannot be issued after the drop operation has been completed. In other words, the CTuner

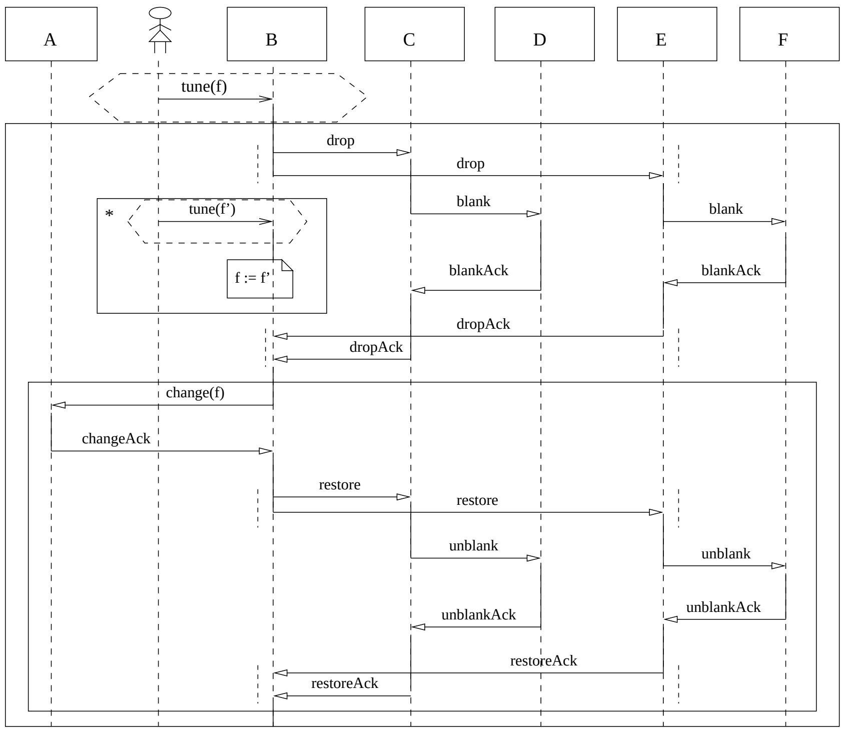

Fig. 4. An LSC for the new configuration

component (component B) does not accept any new $tune(f')$ after it has received *dropAck* from the CVideo component (component C). It can been seen in this LSC that all communications are going according to plan until B receives the very last $tune(f')$. After this message occurs, the main body of the chart is entered where the hot false condition causes immediate (and abnormal) abortion of of the whole chart. In this sense, the LSC shown in Figure 3 describes an anti-scenario of the tuning scenario described in the previous section.

3 Changing the configuration

As mentioned before, the topology of the signal flow in hardware, and consequently the configuration of software components that control individual hardware devices, is likely to change for new products but also during development of a single product. Suppose that a new product has an additional feature for displaying a small screen within the original TV screen. This requires the addition of a Picture-In-Picture (PIP) module for producing a downscaled image which is fed to the video output processor to be superimposed on the existing image. It is a product requirement that the PIP output should be blanked when changing the frequency of the tuner. The PIP samples the image and stores it in an internal memory before the tuner output may be dropped. Like other downstream devices in an analogue television it actually needs some time to complete this operation.

To accommodate the new functionality, the configuration is changed by adding an intelligent PIP component E and its corresponding PIP driver component F. The intelligent component E should be connected to B to be notified when to conduct its blanking operation. By careful examination of the existing component interactions modelled in the LSC of Figure 2 it can be seen that the PIP component's intended part in the new configuration is similar to C's part in the existing configuration.

In fact, B issues drop requests to both C and E in any order. This situation is expressed in the LSC of Figure 4, using *coregions*. In the seminal paper on LSCs [2], coregions correspond to regions in the chart where the events are unordered. They are annotated with dashed vertical lines around the events corresponding to the messages sent. In [5], coregions are given a true-concurrent interpretation: events in the region occur 'at the same time' rather than 'in any order'. In such a case, coregions are denoted by dotted vertical lines around the corresponding events. In our example, and since B issues drop requests in any order, we use coregions in the sense of [2]. Thus the dashed vertical lines around the drop messages sent by B.

Once C and E receive the drop requests, they proceed to request the blanking of D and F, respectively. The two blanking operations are going on concurrently and also, possibly, concurrently with the activity inside the loop too. After the blanking operations have been completed, C and E can drop their signals and return *dropAck* to component B, as before (LSC of Figure 2). The order in which the corresponding drop acknowledgements are received is also irrelevant. The important thing is that B keeps count of the drop requests and knows to expect the same number of drop acknowledgements.

As shown in Figure 4, we use a subchart around the communications that follow the receiving of both drop acknowledgements by B. This has the effect of synchronising components B, C and E and forbidding further progress of the scenario unless B receives both drop acknowledgements. Notice that B's lifeline becomes solid only after both drop acknowledgements have been received, and consequently it only then forces the system to enter the subchart. What happens in the subchart should be straightforward by now.

4 Interpreting scenarios of component interactions

In this section we present the foundations of our approach towards formalising component interactions that appear in a given LSC. Our intention is not to give a formal semantics to LSCs. This is adequately done elsewhere e.g. [2,4,5]. What we propose is to consider LSCs as a starting point for a behavioural model for components [7,8,12]. In fact, the approach presented here has been influenced by the set-theoretic framework for components of [7,8,12].

A few words are in order to clarify some underlying assumptions. First, we deal with synchronous messsages and therefore the send event and the receive event are considered simultaneous. This is consistent with the view taken of synchronous messages in [4,5] and [14] and is indeed the case for message exchange in our example. Second, being focused on modelling component interactions necessary to carry out a specific task, we do not address the prechart or the acts of entering the prechart, exiting it or reaching the end of the main body. We focus on actual system events of sending and receiving messages, what is referred to as *visible* events in [4]. For instance, the assignment of the new frequency f' to the variable f is not considered an event. Incorporating hidden events in future should not prove very difficult.

We begin by considering the graphical picture of an LSC L and attempt to capture its static characteristics with the following definition. Let M_L be the set of messages exchanged in L.

Definition 4.1 We define the body *sort* of L to be a tuple $\Sigma_L = (C_L, E_L, \beta_L)$ where

- C_L is the set of components in L
- E_L is the set of events in L defined as $E_L = M_L \times \{?, !\}$, where $m?$, $m!$ denote the receipt and the sending, respectively, of message $m \in M_L$
- $\beta_L(c)$ is the set of messages that are associated with component c. Function β_L is defined as $\beta_L : C_L \rightarrow \wp(M_L)$.

Note that we use different sets for messages and events: an event here is understood to be the sending or the receipt of a message. In a particular scenario each component will send and receive a collection of messages. Thus, the behaviour modelled in the body of an LSC as a whole may be described by assigning to each component c a sequence $x \in \beta_L(c)^*$, where $\beta_L(c)^*$ denotes the set of all finite sequences over $\beta_L(c)$. This motivates the following definition.

Definition 4.2 Suppose that Σ_L is the sort of an LSC body L. We define V_{Σ_L} to be the set of all functions $\underline{v} : C_L \rightarrow E_L$ such that $\underline{v}(c) \in \beta_L(c)^*$. We shall refer to elements of V_{Σ_L} as *L-vectors*.

Thus, V_{Σ_L} is essentially the Cartesian product of the sets $\beta_L(c)^*$. The interested reader is referred to [12] for the mathematics of L-vectors. The function $\underline{v}$ returns the finite sequence of messages entering or leaving component c, and for each component participating in the interaction described in L. Putting together such sequences, one for each component, we form (a set of) vectors of sequences, where each coordinate corresponds to a component and contains a finite sequence of messages sent or received by that component.

Based on these definitions, an LSC body L can be formally described as consisting of the static structure described by a sort Σ_L together with a language of L-vectors.

Definition 4.3 An LSC body L is a pair (Σ_L, S) where

- Σ_L is the sort of L
- $S \subseteq V_{\Sigma_L}$ is the set of *snapshots* of L

The term *snapshots* refers to a subset of all possible L-vectors, namely those ones that indeed correspond to mandatory behaviour described in L. Intuitively, a snapshot of a system monitored by an LSC L represents behaviour of the system that arises when picking the current location (or graphical position along the lifeline) of each of L's components at any time during a period of activation. The current location of each component indicates which events of that component have already occurred.

The ordering of snapshots is based on prefix ordering of their sequences. First, let us establish our notation. If x and y are sequences, we write $x.y$ for the concatenation of x and y. As is well known, this operation is associative with identity Λ, where Λ denotes the empty sequence. We have a partial order on sequences given by $x \leq y$ if and only if there exists z such that $x.z = y$, and this partial order has a bottom element Λ. Finally, concatenation is cancellative and thus the sequence z is unique.

Ordering among sequences of events in a snapshot maps onto the top-down ordering of the graphical positions (or locations) in which these events occur on the chart. In what follows we use the tuning example presented earlier to illustrate how interactions within that scenario can be interpreted in our formal model. For the LSC of Figure 2 we have $C_L = \{A, B, C, D\}$ and $M_L = \{d, d_A, b, b_A, c, c_A, r, r_A, u, u_A\}$, where we have abbreviated all messages by their first letter and the corresponding acknowledgements with a subscript A to increase readability. The sets of messages associated with each component are given by

$$\beta_L(A) = \{c, c_A\}$$
$$\beta_L(B) = \{d, d_A, r, r_A, c, c_A\}$$

$\beta_L(C) = \{d, d_A, b, b_A, r, r_A, u, u_A\}$
$\beta_L(D) = \{b, b_A, u, u_A\}$

In this case, the set of snapshots for the change frequency LSC of Figure 2 is formed of the following vectors.

$$\begin{gathered} S = \{(\Lambda, \Lambda, \Lambda, \Lambda), \\ (\Lambda, d!, d?, \Lambda), \\ (\Lambda, d!, d?b!, b?), \\ (\Lambda, d!, d?b!b_A?, b?b_A!), \\ (\Lambda, d!d_A?, d?b!b_A?d_A!, b?b_A!), \\ (c?, d!d_A?c!, d?b!b_A?d_A!, b?b_A!), \\ (c?c_A!, d!d_A?c!c_A?, d?b!b_A?d_A!, b?b_A!), \\ (c?c_A!, d!d_A?c!c_A?r!, d?b!b_A?d_A!r?, b?b_A!), \\ (c?c_A!, d!d_A?c!c_A?r!, d?b!b_A?d_A!r?u!, b?b_A!u?), \\ (c?c_A!, d!d_A?c!c_A?r!, d?b!b_A?d_A!r?u!u_A?, b?b_A!u?u_A!), \\ (c?c_A!, d!d_A?c!c_A?r!r_A?, d?b!b_A?d_A!r?u!u_A?r_A?, b?b_A!u?u_A!)\} \end{gathered}$$

In further explanation of the notation, we start with the empty snapshot, which describes minimal behaviour, and gradually build the set of snapshots by including a pair of simultaneous events in each step. Remember that simultaneous events here are considered to be the sending and the receiving of the same message (e.g. $d!$ and $d?$ on B and C resp. in the second snapshot). The last or maximal snapshot describes all interactions that must have taken place in executing the scenario of changing the frequency of the tuner. The ordering among interactions at different coordinates is determined by considering all the predecessors of the maximal snapshot. Considering the skeleton automaton of a main chart described in [2], we may say that it transitions from 'active' to 'aborted' if *any* message occurs out of its sequence given in S. If *all* message occurrences respect the sequencing in S, then the chart will reach the state 'terminated' thus indicating successful completion of the scenario described in L.

The set of snapshots essentially describes the input / ouput behaviour of the participating components and provides additional behavioural information about their state. Each component enters an 'intermediate' state whenever it issues a request $m!$ and returns to a 'stable' state whenever it receives the corresponding acknowledgement $m_A?$. In its stable state, a component guarantees the services it provides. This is not the case though when it is in its intermediate state and thus, in our example, all components do not accept any message (operation call) while in their intermediate state (and assuming one thread of control per component). The only exception is B. This component has an additional 'tolerant' intermediate state. It enters this state whenever it engages in a drop request with C. In its tolerant intermediate state B can accept a new messsage $tune(f')$ and stores the new frequency value for later use. Receipt of *dropAck* brings it back to its stable state. This reflects the typical requirement in CE products, that the last user request is to be served first. These ideas are formally put in the following definition.

Definition 4.4 Suppose that $L = (\Sigma_L, S)$ is an LSC body and $s = (s_1, ..., s_n)$, $s \in S$ is a snapshot of the n components in L, and $c \in C_L$ is a component in L and s_i is the corresponding coordinate in s. We shall say that c is in a *stable* state iff for each $m! \in E_L, m \in M_L$ that appears in s_i, there exists $m_A? \in E_L, m_A \in M_L$ such that $m_A?$ appears in s_i.

If for each $s_i, i = 1..n$, the corresponding components are in a stable state, then we shall say that the snapshot s is in a *stable* state.

The stable state for the receiver component is defined dually. The formal interpretation of the component interactions in the change frequency LSC (see Figure 2) essentially determines which components are in an intermediate state and which are in a stable state, at any time in the course of behaviour exhibited when changing the frequency of the tuner. Whenever a component is in a stable state it also knows the state of the sender component. Note that using OCL 2.0 notation on the message label, as proposed in [6], it is possible to include the sender's name as well. Such behavioural information becomes particularly useful in the face of activities such as adding new components, removing, replacing, versioning of components and so on.

As a small note here, we return to the discussion on hidden events. We could have included an event $t?$ for $tune(f)$ as a first element of the sequence in the second coordinate of all snapshots (except for the empty snapshot). Then we could claim that the prechart is also being considered. In similar fashion, adding a $t_A!$ to the end of the sequence in the second coordinate of the last snapshot would perhaps account for incorporating the act of reaching the end of the main body. However, since we want to model what happens within our component configuration in response to an 'external' event we choose not to include hidden events. In any case, this does not severely affect the essence of our explanations.

5 Conclusions and Future Work

We have presented a formal approach for describing behaviour at the interfaces between components, in the context of a given LSC. Our approach was illustrated by means of a small example from embedded software for consumer electronics products. Component interactions were first modelled using LSCs and then formalised using snapshots generated from LSCs. In addition, an underlying objective was to use the mandatory behaviour described in an LSC as the starting point for our component model described elsewhere [7,8].

Another option for modelling interactions in a component setting would be to use UML sequence diagrams. Interestingly, sequence diagrams in UML 2.0 [10] are undergoing several changes in an attempt to adopt basic concepts from LSCs. In fact, they provide a notion of mandatory behaviour similar to that of LSCs. The main reason for choosing to use LSCs instead is, however, that sequence diagrams in UML 2.0 are still given an interleaving semantics, as in UML 1.x. The situation where two events occur exactly at the same time is common in component-based sytems and therefore a true-concurrent semantics is more suitable. The formal model we are considering [7,8,12] is a model of true concurrency and LSCs are well-suited for a true-concurrent semantics, as shown in [5].

Our approach towards obtaining state information suffers from the inherent assumption that components communicate using the primitive communication mechanism of our example, where a component cannot whatsoever refuse to comply with a request. On the other hand, this seems to be consistent with the view taken in [13] and elsewhere about component contracts as well as the *optimistic* view of component interactions taken in [3]. Specifically, in the horizontal communication style proposed in [14], where components have exclusive ports / interfaces for each of their neighbours, our modelling approach allows a component to perform its part properly without requiring knowledge of the particular configuration. Indeed, each component knows the set of signals associated with each of its ports and thus, by receiving requests or acknowledgements from its closest neighbours it becomes aware of their state.

The formal model, to the extent it has been adopted for interpreting component interactions in a given LSC, and to the extent it is presented here, might seem limited in expressing concurrency and nondeterminism. However, the original component model [7,8,12] is far more expressive. The set out of the model is quite close to the algebraic specification model of [1]. In the sequel however, the two models diverge since our model is based on the order theoretic structure of the set of behaviours of a component and it can, under certain conditions described in [8], be associated to behavioural presentations [11] which are reminiscent of the event structures model of [9]. Thus, we end up with a powerful model of true-concurrency where nondeterminism, concurrency as well as simultaneity can be expressed as distinct phenomena.

The small extension of our example, given in Section 3, corresponds to the use of a fork in [14] which connects one output of a component to two (or more) inputs of other components. A fork in [14] issues requests in any order rather than concurrently. Further, a switch is used for connecting one of n inputs to one output. The use of such connectors accounts for parallelism, and/or concurrency, and mutual exclusion of issued requests. It would be interesting to see how the conditions that enable the association to behavioural presentations can be imposed on the order theoretic structure of snapshots. This would allow us to use behavioural presentations to determine the temporal relations among occurrences of events appearing in the snapshots.

One other possible extension of the work presented here has to do with associating snapshots with automata. Based on consequences of the conditions that allow the association to behavioural presentations, a component can be associated with a certain class of automata [12]. In this paper, we considered specific scenarios, described in LSCs, as the starting point for modelling component behaviour. In a certain important sense, we restricted the environment to mandatory behaviour only. This view is consistent with the *optimistic* view taken in [3]. It would be interesting to further investigate the effect of a constrained environment on the automata derived from our model.

Acknowledgement

We would like to thank the anonymous referees for their valuable comments.

References

[1] M. Broy. Algebraic Specification of Reactive Systems. *Theoretical Computer Science*, 239(2000):3–40, 2000.

[2] W. Damn and D. Harel. LCSs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[3] L. de Alfaro and T. Henzinger. Interface Automata. In *Foundations of Software Engineering (FSE'01)*, pages 109–120. ACM Press, 2001.

[4] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.

[5] J. Klose and H. Wittke. An Automata-based Interpretation of Live Sequence Charts. In T. Margaria and W. Yi, editors, *TACAS 2001*, volume 2031 of *LNCS*. Springer, 2001.

[6] J. Küster Filipe. Giving Life to Agent Interactions. In H.-D. Ehrich, J.-J. Meyer, and M. Ryan, editors, *Agent, Objects and Features: Structuring mechanisms for contemporary software*, LNCS. Springer, 2004. To appear.

[7] S. Moschoyiannis and M. W. Shields. A Set-Theoretic Framework for Component Composition. *Fundamenta Informaticae*, 59(4):373–396, 2004.

[8] S. Moschoyiannis, M. W. Shields, and J. Küster Filipe. Formalising Well-Behaved Components. In *Proceedings Formal Aspects of Component Software (FACS), satellite workshop of FME'03*, pages 121–142. UNU/IST No. 284, 2003.

[9] M. Nielsen, G. Plotkin, and G. Winskel. Petri Nets, Event Structures and Domains, part 1. *Theoretical Computer Science*, 13:85–108, 1981.

[10] OMG. *UML 2.0 Superstructure Draft Adopted Specification.* OMG document ad/03-01-07, available from http://www.omg.org, August 2003.

[11] M. W. Shields. Behavioural Presentations. In de Bakker, de Roever, and Rozenberg, editors, *Linear Time, Branching Time and Partial Orders in Logics and Models for Concurrency*, volume 354, pages 671–689. Springer Verlag, 1988.

[12] M. W. Shields and D. Pitt. Component-Based Systems I: Theory of a Single Component. Technical Report SCOMP-TC-01-01, University of Surrey, 2001.

[13] C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison Wesley, 1997.

[14] R. van Ommering. Horizontal Communication: a Style to Compose Control Software. *Software: Practice and Experience*, 2003. To appear.

[15] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala Component Model for Consumer Electronics. *IEEE Transactions on Computers*, 33(3):78–85, 2000.

[16] ITU-TS Recommendation Z.120. *Message Sequence Chart (MSC).* ITU-TS, 1996.