The Framework of an Embedded Software Controller using the Transaction Mechanism

Leonardus M. M. Veugen
tis.veugen@gmail.com

Abstract

In a real-time embedded software environment the interfaces of concurrently executing components are specified by (port) protocol state machines. Software controllers are special components needed for the coordination of activities of other components. The state configuration of a controller consisting of the Cartesian product of protocol states, can become obscure due to the interrupting autonomous behaviour of the surrounding components. In this paper we present a framework for a controller that is able to handle the state explosion problem. This framework contains relevant architectural viewpoints and a run-time environment. The design of the controller contains proxy sub-components that shield access to the controlled components. The sequence diagrams of the controller's behaviour are in two steps transformed to superstates, called transactions, that interact with the proxies.

Keywords: software controller, sequence diagram, state machine, transaction

1. Introduction

The complexity of computer systems keeps growing due to unlimited user demands. Fulfilling those demands requires powerful processing power from a variety of CPU cores and network access to servers and peers at the expense of huge software systems controlling all those resources. The divide and conquer principle must be applied to realise a reliable software system. Decomposition of systems into subsystems, and subsystems into components leads to manageable parts for software architects and developers. The price of decomposition is the need for coordination between the activities of all individual components. The special class of components, denoted as software controllers, are aimed for providing such coordination. In this paper we consider a real-time embedded environment where many components run in concurrent threads and communicate asynchronously. The design of the components is supported by finite state machines [1] to pinpoint allowed activities and ongoing activities. The interfaces of components are specified by protocol state machines. Complex software controllers must deal with many interfaces to connected components, and so will have an extensive state configuration based on the Cartesian product of protocol states. A stimulus to the controller activates a scenario that depends on the current state configuration. In an embedded system, components can interrupt a scenario resulting in all kind of unforeseen state configurations. Handling the explosion of configurations is an underestimated topic that only seems to be solved by brute force [6]. Several papers (e.g. [10], [8]) are devoted to the conversion from sequence diagrams (describing scenarios) to finite state machines, but dependencies on protocol states are not taken into account in the sequence diagrams. To solve the state explosion problem we propose an approach, named Transaction Mechanism Methodology, for designing a software controller in a structured manner. This methodology has evolved from several projects during the last decade [9]. Recent insights contributed to a refinement where the architectural aspects of a software system are taken into account.

This paper is organised as follows: section 2 lists the relevant architectural viewpoints guiding the development of the software components. In section 3 a run-time environment is sketched that takes away trivial burden from the developer. Section 4 describes the internal design of a software controller in detail, illustrated with examples. This design introduces transactions, special superstates, that are derived from sequence diagrams describing the controller's behaviour. The controller will contain proxy sub-components that obey the (port) protocol state machines of the controlled components. The transactions access the proxies such that a state explosion from protocol state machines is prevented. Finally, section 5 contains conclusions.

2. Architecture

The architecture of a software system can be described from several viewpoints. As [3] shows, there is no unanimous set of viewpoints satisfying all

architectural needs. In this paper we select our own set of three viewpoints that suit the architecture in the Transaction Mechanism. The views of the viewpoints are specified by means of UML diagrams [7].

Components

Components are the basic elements in an architecture. Therefore, we firstly characterise components and their interactions. A component can be either a leaf component or a compound component consisting of leaf components and/or compound components. A leaf component contains one ore more classes. Components can be instantiated causing the instantiation of its contained elements.

A strict layering hierarchy is maintained in the interaction between components. A client component in a higher layer sends stimuli to a lower layer server component; the stimuli will be realised by function calls. A server component can also send stimuli to a client component, realised by callback functions. Mutual interaction between peer components is allowed: the peers will both fulfil a server role as well as a client role for accepting callback functions. The interaction between components distinguishes between synchronous communication and asynchronous communication. Synchronous communication happens between a server component having a provided interface and a client component having a required interface. On the other hand, asynchronous communication happens between a server component having a provided port and a client component having a required port. In general, a port contains both a provided interface and a required interface. A provided port contains a provided interface servicing calls from a client component, and also a required interface specifying callback functions that must be implemented by that client component. A required port contains a required interface towards a server component, and also a provided interface of callback functions invoked by that server component.

The API of a server component can be specified using protocol state machines, that provide a simple and yet powerful technique for conducting the access to that component. We distinguish between passive and active state behaviour of the API. In case of passive state behaviour, state changes are only caused by stimuli from a client component, realised by synchronous communication via interfaces. The standard UML Protocol State Machine (PSM) indicates allowed transitions at a server component. In case of active state behaviour, a component can itself change state due to a so called *native event*[1]. The actual cause of such a native event inside a component is irrelevant for the protocol specification. A state change due to a native event must be reported by asynchronous communication to a client component. Therefore the interaction point is a port for receiving stimuli and sending stimuli. The port has one single state machine with state transitions due to received stimuli, and state transitions resulting from native events. Since a standard UML PSM does not allow activities (for sending stimuli) an extension of the PSM is needed, called Port Protocol State Machine (PPSM). The required port of the client component has a conjugate version of this (P)PSM [5], with state transitions due to native events causing stimuli sent to the server component and optionally with state transitions due to stimuli received from the server component.

Module Viewpoint

A complex software system is decomposed in many components for practical reasons. In our environment, a repository contains the involved components in a variety of representations. This viewpoint has three views: class view, component view, and package view. The corresponding UML diagrams and their purposes are respectively:

a) Class diagram, for
   - specifying classes, ports and interfaces,
   - indicating relationships between classes, ports and/or interfaces.

b) Component diagram, for
   - identifying the software components together with their dependencies,
   - indicating the interfaces and ports of a component.

c) Package diagram, for
   - grouping components into a package,
   - grouping packages into super-packages,
   - indicating the dependencies between packages,
   - applying namespaces.

Structure Viewpoint

This viewpoint addresses the structure of the system using components available in the repository. This structure lists the components to be instantiated, and also the connections between component instances. Connections can be made from a required interface to a compatible provided interface, or from a required port to a compatible provided port. A structure can have several hierarchical levels reflecting the recursive decomposition starting from the top level context diagram. Interfaces and ports of a compound component must be delegated to the contained sub-components and/or contained classes.

[1] The term *native* is used to prevent confusion with an *internal* transition.

The Structure Viewpoint also shows active (compound) components that execute inside their own thread.

UML composite structure diagrams will be used for describing the system structure.

Behaviour Viewpoint

This viewpoint describes the state dependent behaviour of component instances. It consists of three types of views. Firstly, the protocol view concerns the mutual interaction between two connected components having passive or active state behaviour. In fact, the interaction happens between two connected interaction points (interfaces or ports). The (port) protocol state machines in this view are realised by UML state machine diagrams.

The second view, the design view, concerns the externally observable effects of a component as a consequence of received stimuli. A received stimulus can either be a function called at a provided interface (of a provided port), or a callback function invoked at a provided interface of a required port. The impact of a stimulus depends on the overall state of the affected component. Therefore, the state of each (P)PSM must be modeled in the design view. This view is realised by UML sequence diagrams.

The third view, the implementation view, deals with the internal behaviour of a component, that will be realised by a Finite State Machine (FSM). The FSM results from (P)PSMs and sequence diagrams according to the method described in chapter 4. The realisation of this view hence yields UML state machine diagrams.

3. Run-time environment

A software architecture can still leave much freedom for the implementation of the components in a software system. From the composite structure diagrams a component developer can still ask questions such as:

- Who instantiates my component?
- How are connections realised?
- How are such connections used?
- Does my component need a thread?
- Who cares for threads or thread pools for my instance?

Solving these aspects for each component by each developer individually has several drawbacks: learning curve, non-uniform solutions, sub-optimal approach per component. As an alternative solution we propose the usage of a run-time environment (RTE) taking care of component instantiation and setting up connections [4]. Additionally, our RTE has an extra service of creating threads. The usage of a RTE has advantages such as quality improvement, reusability, flexibility. Each component must provide a mandatory interface on behalf of the RTE. This drawback is compensated by the fact that the highly reusable interfaces clearly identify the obligatory functions. The functions of this mandatory RTE interface are listed in Table 1.

The kind of references mentioned for the connections depend on the used programming language. For object oriented languages the references are objects. A call along a port to a lower layer component is realised by calling a method of the connected object (reference). For functional languages more effort is needed to realise references. In fact the facilities of an object oriented language must be emulated.

Function	Description
Construct	Create an instance (factory function)
Initialise	Internal initialisation actions; no other instances are accessed yet since they might not yet be initialised
GetProvIfc	Return a reference to a provided interface
SetReqIfc	Pass the reference to a required interface
Connect	Connect a provided interface to a required interface.
Prepare	Mutual initialisation actions; connected instances may be invoked
Run	Activate an instance (optional function)
Halt	Deactivate an instance
Cease	Mutual termination action; cleanup actions at connected instances
Disconnect	Disconnect a provided interface from a required interface
Terminate	Internal cleanup actions
Destruct	Delete the instance (factory function)

Table 1 : RTE interface

Figure 1 shows the life cycle of a component instance.

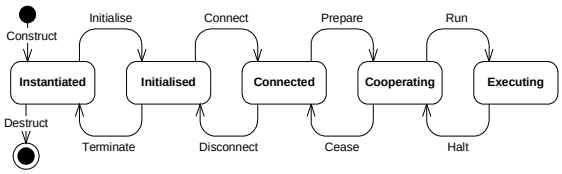

Figure 1 : life cycle of a component instance

The RTE uses the overall composite structure diagram of the software system to assemble and execute the system's application. The components needed for assembling are taken from the repository.

Our RTE has an internal component implementing thread services. The software components themselves do not need to implement threads! A developer need not figure out (portable) thread implementation details. The major benefit, however, is the flexibility for an architect in thread manipulation. The RTE creates a thread instance on behalf of every active component in the Structure Viewpoint. Such a thread instance has provided ports and required ports that hook between the connections of connected components.

Figure 2 shows the situation for an active component S that effectively is accessed as a passive component by the embracing active component T that implements a thread. Component T intercepts all stimuli for S, both the stimuli from the client component U, as well from the server component E. Callback stimuli from E are intercepted to prevent cyclic invocations. Interception can easily be realised because RTE takes care of connecting the interfaces. Function calls and callbacks from S to its environment can be invoked directly without involvement of T .

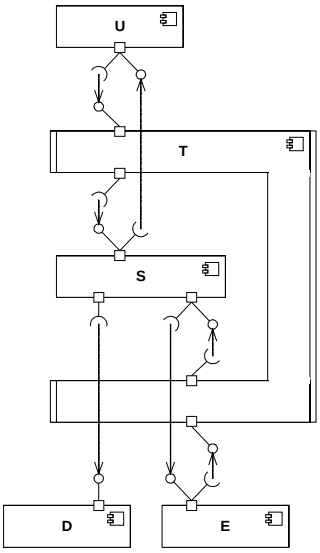

Figure 2 : Structure diagram for active component

A thread component has 2 sub-components: a frontend and a backend. The frontend, running in a caller's thread, accepts a stimulus of a (callback) function, encodes the stimulus in a message containing a stimulus command and the accompanying parameters of that stimulus, and puts the message in a so called external queue. The backend of the thread component, running in its own thread, decodes a dequeued message and executes the call(back) at the embraced component. Hence, inside T wrappers are needed for encoding/decoding functions. For relieving a developer, such wrappers can be generated automatically when the interfaces have been described in a formal way [2].

Function calls from U to a provided port of T require a synchronisation mechanism inside T: the frontend of T has to block the call of client U until the corresponding call to S has returned from the backend. Then, an optional return value can be passed and the call can be unblocked. No synchronisation is needed for callbacks at a required port: the frontend queues the callback message, and the invoker can then continue processing. Messages from different ports can be stored in a single queue or in multiple queues; even priority based queues could be used. Adequate choices for the queues depend on the application's real-time requirements.

Passive components are not assigned to a thread: they execute in the context of a caller's thread. Callbacks of internal sub-components of passive components must execute in some thread. This thread must be assigned statically, and therefore can not be the thread of an arbitrary caller. The RTE will dispose a special thread on behalf of passive components.

An embraced component can be a compound component, as sketched in Figure 3 where S is decomposed in G and H.

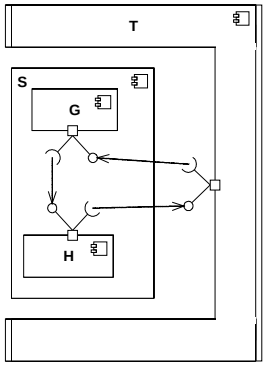

Figure 3 : Interception of callbacks of sub-component

A function call from G to H is executed directly via the connected reference of the provided port of H. However, a callback from H to G is not executed directly to prevent cyclic invocations. Instead, callbacks are encoded into a message and queued in a so called internal queue of component T. The internal queue has a higher priority than the external queue; messages are dequeued from the external queue when the internal queue is empty. Note that again wrapper functions are needed for handling the callback stimuli to G.

4. Design of Software Controller

A Software Controller (SWC) is a component that controls several other server components, named Functional Components (FCs), at a lower layer [5]. The SWC accepts calls from a client component, named User Component (UC), interprets these calls depending on its state machine, and invokes the Functional Components accordingly. The processing of the FCs can take some time, forcing the communication to be asynchronous. As a consequence the SWC is an active component. In general, the communication between UC and SWC will also be asynchronous. The SWC obeys the rules of the above architecture, and it will execute under supervision of the RTE.

A good starting point for describing the behaviour of a SWC are Sequence Diagrams (SDs) that are initiated by user requests from the UC. Such SDs end with a stable state configuration, being a well defined starting point for modeling another user initiated Sequence Diagram. SDs in a real-time system often have asynchronous behaviour. Hence, a SWC issuing an asynchronous request to a FC must wait for a response from that FC. During the waiting interval the SWC can be subject to stimuli that do not fit in the currently executing SD. For example, a user might change her mind and wants to overrule the last choice leading to another user request. System complexity increases even more when FCs can issue autonomous requests, e.g., in a DVD player requests may happen due to disc read errors, resource problems, etc. These requests are often so demanding, unlike user requests, that the current SD must be aborted, and replaced by a new SD for serving the autonomous request. The major problem is that the SWC is interrupted in an arbitrary state configuration, consisting of the states of all its (P)PSMs and its own internal state. The new SD must be able to cope with all those arbitrary state configurations. In fact, for a single stimulus many different SDs must be modeled, one for each state configuration. This means not only a lot of work, there is also the question whether an architect is able to figure out all the possible configurations. Missing a single state configuration may already lead to an unreliable system. This phenomenon of state explosion imposes not only solid craftsmanship of an architect, it requires also good skills in coding and testing. However, instead of relying solely on the capacities of the software development team, it would be better to apply a method that simplifies this problem.

We propose a method that reduces the impact of the state explosion. The basic idea is that in a SD each functional component has a certain target state. The weak assumption is made that this target state can be reached from the current state, otherwise the component's state machine would be incomplete. The controller can not ask the FC to enter the target state directly because of possible protocol violations. The solution is to construct for each FC a proxy component shielding the access to that FC. The proxy components are designed as sub-components of the SWC, and hence hidden for the controller's environment.

Figure 4 shows the composite structure diagram of a SWC with 3 Functional Components named D, E and F. The User Component U is connected via PPortU. In this example, the Functional Components have different natures: component D, connected via RPortD, has no state behaviour; component E, connected via RPortE, has passive state behaviour; component F, connected via RPortF, has active state behaviour.

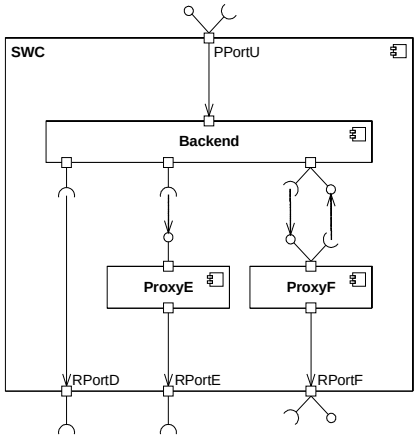

Figure 4 : Decomposition of SWC

Sub-component Backend takes care of the processing of the SDs. ProxyE is the sub-component that shields the access to Functional Component E. Similarly, ProxyF shields access to F. The figure illustrates how the interfaces and ports of the controller are delegated to the sub-components. The provided port of the SWC is realised by Backend; required interface RPortE is realised by ProxyE, and required port RPortF is realised by ProxyF. The Backend is not connected directly to a FC with state behaviour, but instead to its proxy. The provided interface/port of a proxy has to resemble the provided interface/port of the corresponding FC. Each function of a provided interface of a FC will have an equivalent function at the provided interface of its corresponding proxy. Stimuli from the required interface at a proxy's provided port will be similar to those from the corresponding FC's provided port. Each proxy has a reporting facility telling whether the target state has been reached. It realises this facility

by exporting an additional boolean function IsProxyOk(), returning *true* in the target state. Note that this inquiry function reflects the last call to a proxy.

Firstly, the working of a proxy with active state behaviour is explained, by means of the following example. It shows how the PPSM at the provided port of a FC transforms to the PPSM at the provided port of its proxy. The introduced stereotype «ProxyOk» indicates that the target state has been reached, whereas «ProxyNok» tells that it has not yet been reached.

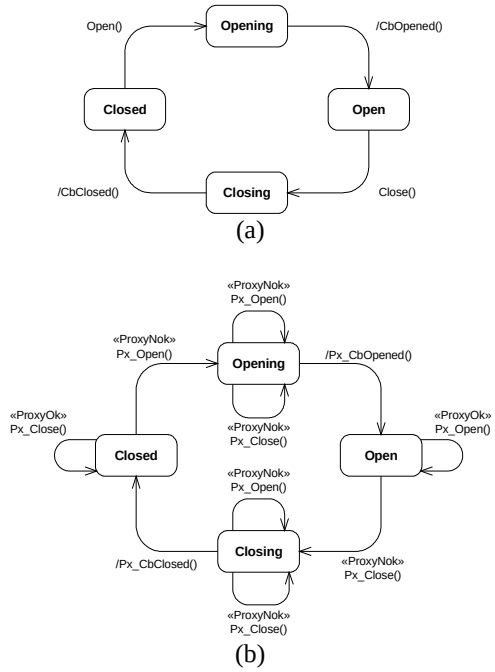

Figure 5 : (a) PPSM of FC, (b) transformed PPSM of Proxy

Figure 5a shows the PPSM for a resource that can be opened and closed. Note that in state Opening the call Close() is forbidden. Figure 5b shows the transformed PPSM at the corresponding provided port of the proxy. Based on the latter PPSM the Backend can make a valid Px_Close() call in state Opening.

The SD in Figure 6 shows how the 3 involved sub-components Backend, Proxy and FC manage to successfully execute the Close() request. The Backend keeps making a call until the Proxy tells that the target state has been reached. Every (expected) response event triggers the Backend for repeating such a request. In fact, the actual response event does not matter; even response events from other Proxies, may activate the Backend repeating its request. The only reason for distinguishing between the response events are the optional accompanying parameters. If all such parameters could be fully handled in the Proxy components then the Proxies could raise a uniform trigger event to the Backend instead of the original response events.

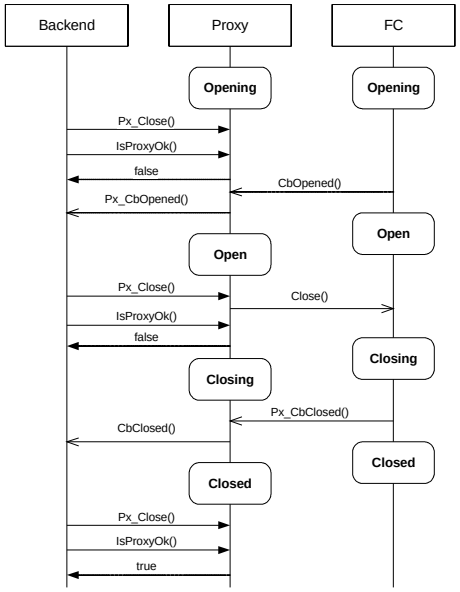

Figure 6 : Sequence Diagram of Close() request

In the shown example no function parameters are needed as is often the case for concluding activities. If function parameters must be passed to an FC, then they can be passed directly from the Backend to the Proxy or the Proxy should be able to fetch them indirectly.

The situation of a Proxy with passive state behaviour is easier than described above for active state behaviour. The Proxy can translate the proxy function call to one or more synchronous calls that are invoked consecutively at the FC.

A Proxy is memoryless: it does not remember that a previous request has been issued. This design choice is made on purpose for the following reason. Suppose that a SD driving an Open() call would be interrupted by a SD that wants the target state to be Closed (by means of a Close() call). In these circumstances, the Open() call has become obsolete, and storing it in some state variable would make the Proxy component needlessly complicated.

Design of Backend

The Backend is the main processing unit for the execution of sequences modeled by the SDs. Since the Backend communicates with the Proxies and not with the FCs, the SDs of the SWC can not be applied by the Backend. Backend SDs must be constructed from the original SDs of the SWC. The interaction between User Component and Backend can be copied

from the SDs of the SWC because of the equivalence of the ports. The interaction between the Backend and the Proxies resembles the interaction between SWC and FCs. However, an essential difference is that each asynchronous function call is replaced by a synchronous function call and a call to the target state inquiry function. The replaced function is called synchronous because it need not wait for a specific asynchronous response. In fact, the synchronous function has a polling behaviour. From a performance perspective, this kind of polling is efficient since it only happens after a received response, unlike polling at fixed time intervals.

Figure 7 shows a fragment of an SD with asynchronous request-response interaction between Backend and FC. At the beginning of the loop the target state is checked. If IsProxyOk() returns true then the fragment has finished. Otherwise, the Backend waits for any response event, repeats its request, and again inquires IsProxyOk().

The SD transformations for the Backend are an intermediate step towards the implementation of the Backend. The next step is a conversion from SDs to FSM. The reason for such a transformation is that FSMs, unlike SDs, are well suited for generating source code. Our architecture leads to another good reason for such a conversion. The purpose of (port) protocol state machines is mainly to specify interfaces. For the implementation of the Backend a FSM must be constructed. The PPSM of the PPortU is used as skeleton for this FSM. In the sequel we show how this skeleton is extended with SD based FSMs.

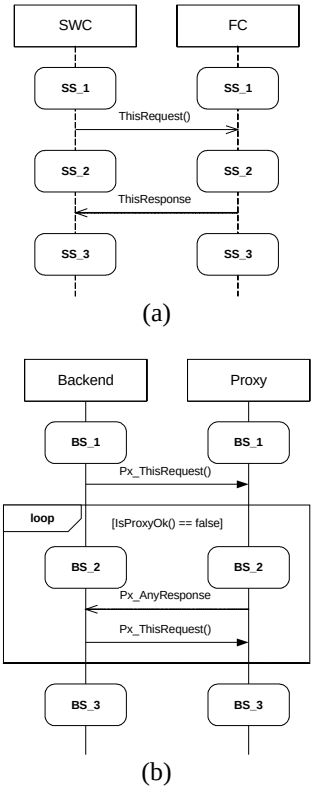

Figure 7 : Primitive sequence diagrams, for (a) SWC and (b) Backend

Transactions

We discern four typical SD patterns, as sketched in Figure 8. For all patterns the SD references contain any sequence of synchronous and asynchronous interaction between SWC and its FCs.

In Figure 8a the UC makes an asynchronous request and after interaction controlled by the SWC a response is sent to the UC. In Figure 8b the UC makes a synchronous call, and the SWC starts processing this call by interacting with the FCs. The UC is blocked during this interaction. Finally the UC is unblocked and the synchronous call returns (optionally with return parameters and/or return value). In Figure 8c a FC reports an autonomous event that after optional interaction is forwarded to the UC (e.g., a recording ends due to "disc full"). In Figure 8d a FC reports an autonomous event that can be handled internally by the SWC (e.g., during playback from an optical disc a read error is recovered by skipping several disc sectors).

Each SD fragment can be transformed in a state

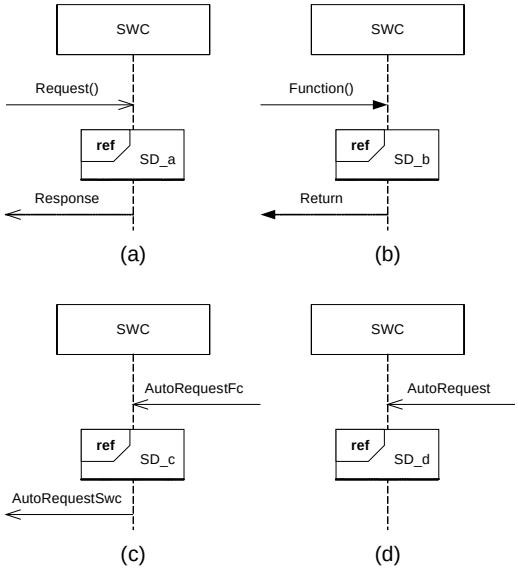

Figure 8 : Typical Sequence Diagram patterns

machine, called transaction. Such a transaction is a superstate containing a sequence of transaction states accompanied with activities, refer to Figure 9 for a primitive version of a transaction.

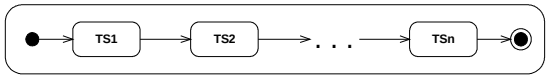

Figure 9 : Primitive transaction

A transaction state is a small superstate implementing the polling behaviour as outlined in the sequence diagram of Figure 7b. The conversion from this sequence diagram to a template state machine is shown in Figure 10.

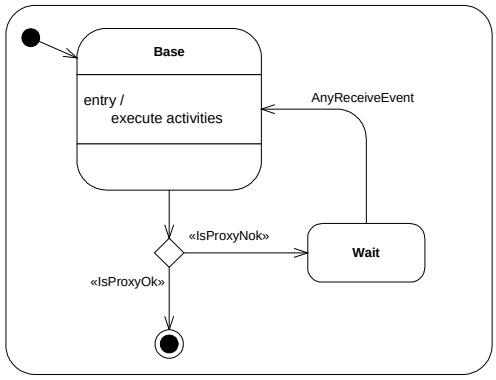

Figure 10 : Transaction state template

At the choice state (with the diamond shape) the target state is checked. The stereotype «IsProxyOk» represents the guard: IsProxyOk() == true, whereas «IsProxyNok» represents the guard: IsProxyOk() == false. The transition from state Wait to Base is triggered by any event as indicated in the figure by the UML AnyReceiveEvent. It means that the transition is executed for any trigger offered to the state machine, unless there is a transition from the source state or from a superstate that concretely matches the trigger. This concept can be considered as the 'else' case when selecting a firing transition.

In practice, a SD fragment can be complex with branches or even loops. But even then a transaction can be constructed by means of basic elements of transaction states or compound elements of sub-transactions.

In the subsequent step, the transactions are integrated with the PPSM of the UC. The 4 SD patterns of Figure 8 are worked out by means of examples:

Ad a. The PPSM between UC and SWC exhibits a request-response interaction. After a request the user will wait in an intermediate state for the controller's response. The SWC moves to the mirrored intermediate state and executes its activities there. Figure 11 shows this interaction as a state machine

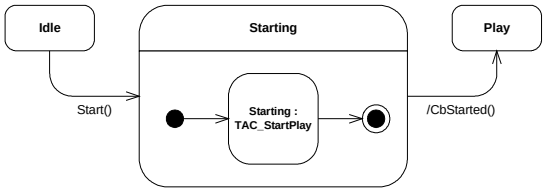

Figure 11 : Transaction due to request-response interaction

fragment. The Starting state has become a superstate containing the TAC_StartPlay transaction. At the end of the transaction an automatic state change happens to state Play issuing the CbStarted() response.

Ad b. The user calls a function, and remains blocked until the SWC causes the function to return. A typical example is a function that retrieves some information. The SWC might need access to its FCs asynchronously, and so requires a transaction. According to the transaction state template, the Play superstate in Figure 12 contains a Base state, but now extended with a transaction. The GetInfo() call is a stimulus causing a transition inside the Play state. This is realised by a transition from the Base state to the TAC_GetInfo transaction, and then back to the Base state.

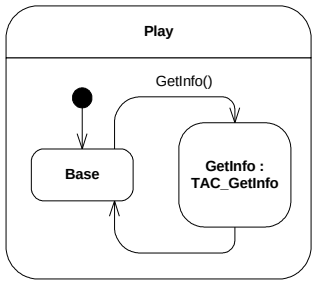

Figure 12 : Transaction due to function call

Ad c/d. Autonomous events (or also timer events) may be the cause of starting a transaction. For example, a read error during playback of an optical disc may be recovered by skipping several disc sectors. Figure 13 shows a CbReadError() stimulus raised by a FC. Firstly, a transaction TAC_Stopping is executed to prevent any unwanted visible or audible artifacts. Then, a test is performed about the number of recent read errors. If there are too many errors, then the Play state is left via the ExitError point. As a consequence playback stops and the SWC will transition to an Error state, indicating this by a

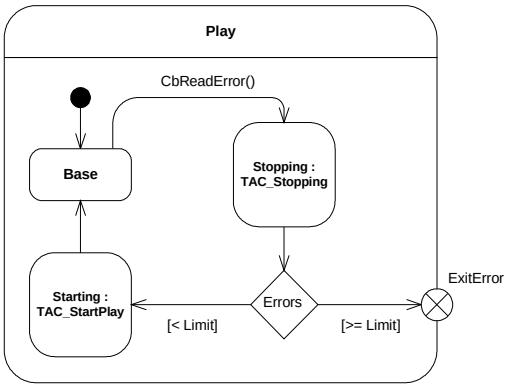

Figure 13 : Transaction due to autonomous event

CbError() event (reflecting SD_c). Otherwise, playback is restarted via the TAC_StartPlay transaction, and afterwards a transition to the Base state is made (reflecting SD_d).

The latter case shows how transactions can be modeled as reusable parts:

- the TAC_StartPlay transaction is also used when starting playback from Idle
- the TAC_Stopping transaction is also used when the user stops playback

The above basic examples illustrate that the transactions are building blocks in an FSM. A single transaction can be applied multiple times saving development effort. Transactions apply the concept of hierarchical state machines keeping FSMs readable and maintainable.

More complicated situations may arise, when a user event or an autonomous event happens during a transaction. A possible solution for handling an interrupting user event is to defer that event, at the expense of possible undesirable observable effects. This solution should be feasible for normal user events, such as a Pause() call during playback, because a user may wait for response without harming the system. An alternative solution would be to design a new combined transaction with the drawback of increasing the complexity of an FSM. But, the most elegant solution would be to process the interrupting event and then continue with the execution of the current transaction. We will proof the feasibility of such a solution by means of the following use case.

Suppose, playback of an optical (video) disc is hampered by a read error forcing the system to recover internally. At an arbitrary moment during this recovery the user wants to pause the playback without knowing or noticing the recovery activities. The relevant (simplified) parts of the FSM are shown in Figure 14. Main state Play contains in substate Recovering the transaction for the recovery of a read error. This TAC_Recover transaction is followed by a transaction to restart playback with normal play speed. Main state Pausing also contains the TAC_Recover transaction, that is followed by a transaction to present a new paused video frame.

If the trigger Pause() is given in substate Play.Recovering, then the trigger can be honoured by just transitioning to substate Pausing.Recovering . Since the transactions of the old and the new substate are identical, there is no need to execute TAC_Recover from scratch. The current (deep) substate inside the TAC_Recover transaction can be inherited from main state Play to main state Pausing. We introduce the stereotype «TacTransition» for a transition between 2 transaction states to allow for substate inheritance. The precise interpretation is that after the transition from a source superstate to another identical target superstate the exact substate configuration inside the target superstate equals the substate configuration of the source superstate.

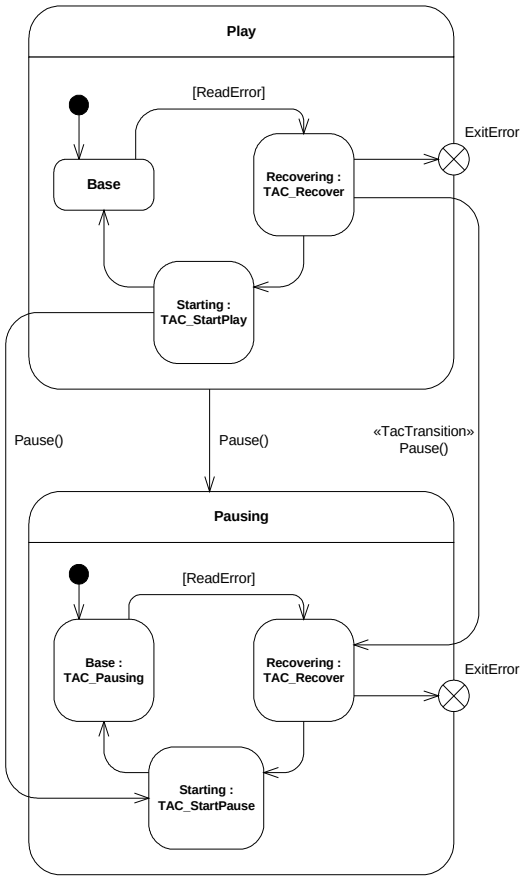

Figure 14 : Transition variations from *Play* to *Pausing* by stimulus Pause()

If the trigger Pause() is given in substate Play.Starting, then the transition results in entering Pausing.Starting and starting transaction TAC_StartPause.

5. Conclusions

This whitepaper presents an integrated development approach that evolved from designing software controllers in an embedded real-time system environment in the course of several projects during the last decade. The gained practical experiences and the applied theoretical developments in the computer science area have resulted in the presented approach. It encompasses a framework consisting of architectural viewpoints, techniques, tools and reused software for developing software components and controllers in particular. A software controller has many interfaces to its upper layer user component(s) and its lower layer functional components. The behaviour of a controller is an architectural task because of the interactions to its surrounding components. The relevant, involved architectural viewpoints have been treated to guide architects and developers. A software controller is a complex component containing sub-components to which the architectural viewpoints can be applied.

The interfaces of a controller are specified by (port) protocol state machines accounting for passive and active state behaviour at these interfaces. The state explosion due to the Cartesian product of the states of the (port) protocol state machines of all interfaces has been prevented by the introduction of proxy sub-component inside a controller. Each proxy shields the access to its corresponding functional component in a robust way. As a consequence, the sequence diagrams describing the inter-component behaviour with respect to the controller have become tolerant for the possible states of the functional components.

The sequence diagrams can be converted to transactions with transaction states in an intuitive manner. The internal overall state machine of the controller can be constructed in a straightforward manner from the port protocol state machine of the user component by adding the main transactions of the controller's sequence diagrams. Furthermore, the state machine diagram has been enriched with a new transition concept. The transition with stereotype «TacTransition» enables an efficient implementation of a transition between 2 identical transactions preserving the substate(s).

References

[1] David Harel, *Statecharts: A Visual Formalism for Complex Systems*, Science of Computer Programming 8, 1987, pp. 231-274

[2] David Alex Lamb, *IDL: sharing intermediate representations*, ACM Transactions on Programming Languages and Systems, Volume 9, Issue 3, pp. 297-318 (July 1987)

[3] Nicholas May, *A survey of software architecture viewpoint models*, in J.-G. Schneider, ed., 'The Sixth Australasian Workshop on Software and System Architectures (AWSA 2005)', Swinburne University of Technology, Melbourne, Australia, 2005, pp. 13–24, [ISO/IEC 10746]

[4] *Robust Open Component Based Software Architecture for Configurable Devices Project*, Deliverable 1.5, July 2003, http://www.hitech-projects.com/euprojects/robocop

[5] Bran Selic and Jim Rumbaugh, *Using UML for modeling complex real-time systems*, Technical report, ObjecTime, March 11, 1998

[6] Osamu Shigo, Atsushi Okawa, Daiki Kato, *Constructing Behavioral State Machine using Interface Protocol Specification*, 13th Asia Pacific Software Engineering Conference, APSEC 2006, pp. 191-198

[7] *OMG Unified Modeling Language*, November 2007, http://www.omg.org

[8] Simona Vasilache and Jiro Tanaka, *Synthesis of State Machines from Multiple Interrelated Scenarios Using Dependency Diagrams*, Journal of Systemics, Cybernetics and Informatics, Vol.3, No.3, 2006, pp. 18-25

[9] Tis Veugen, Dave Boshoven, *Transaction Mechanism Methodology for an Embedded Software Controller*, Philips Applied Technologies, internal report, 2004-10-21

[10] Tewfik Ziadi, Loïc Hélouët, Jean-Marc Jézéquel, *Revisiting Statechart Synthesis with an Algebraic Approach*, Proceedings of the 26th International Conference on Software Engineering 2004, pp. 242-251