From Use Cases to System Implementation: Statechart Based Co-design

Luís Gomes, Anikó Costa
Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia, Dep. of Elect. Eng.
& UNINOVA, Centro de Robótica Inteligente
2825 Monte de Caparica, Portugal
{lugo, akc}@uninova.pt

Abstract

This paper proposes a methodology for embedded systems co-design, based on statechart models. The process starts with grabbing the system functionalities through use cases. A set of procedures addressing the implementation of Statechart models is presented. The main goal of this set of procedures is to lift the structuring mechanisms presented in statecharts to the top level. In this sense, the complexity of statechart implementation will be similar to the complexity of communicating concurrent state machines and the platforms selected to support implementation will not need to have specific capabilities to directly support the structuring mechanisms of Harel's statecharts. As a consequence, full direct implementation of statecharts is possible considering different types of implementation platforms, ranging from hardware-centric or software-centric to hardware-software partitioning through co-design techniques.

1. Introduction

The concurrent design of hardware and software proved to be effective to improve cost-performance relation in the design of embedded systems when compared with the traditional way of separated design flows associated with hardware and software components.

Many computational models have been referred in the literature to specify embedded systems, accommodating co-design techniques.

In the present paper, we may roughly characterise an embedded system through the following equation:

Embedded system =
Reactive system
\+ Real-time constraints
\+ Data processing capabilities (1)

In this sense, the selection of a model of computation should consider the reactive nature of the embedded system, without forgotten real-time and data processing constraints.

From the point of view of the reactive system, ideally, a model of computation should include capabilities to represent concurrency and sequential behaviours, and to assure communication and synchronisation among concurrent components.

Also, accommodating hierarchical model structuring mechanisms is a key convenience, from the engineering point of view, in order to enable a compact representation of the system model using different levels of abstraction.

In the present paper, we propose a development methodology applicable for embedded system co-design, starting from high-level requirements, characterised in a very informal way, and ending-up with the implementation code, that is adequate to be used in different platforms, ranging from software-centric to hardware-oriented components.

The paper starts with a brief description of the proposed methodology and characterisation of the statechart formalism. In the following section, an example is introduced in order to allow illustration of the referred procedures. Co-design specificities come into action in the next sections through the analysis of the model partitioning and statechart implementation issues, considering either hardware, software or mixed based implementation platforms. At the end, final remarks regarding the example is produced and conclusions are presented.

2. Methodology overview

According to [1], the specification of a system should accommodate several views, namely:

- Functional view; this view is responsible to capture "what" should be implemented, in terms of inputs and outputs, functionalities and activities;
- Behavioural view; this view is responsible to capture "when" should the system exhibit some specific behaviour or take some specific action; it is responsible to model the dynamics of the system, in terms of the control point of view and from the time dependency of events; aspects like concurrency and synchronisation should also find a way to be represented;

Proceedings of the First ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE'03) ISBN 0-7695-1923-7/03 $17.00 © 2003 IEEE

- Structural view; this view is responsible to capture "how" should the system be composed in terms of the components and interconnections between them.

Also, according to [2], non-trivial systems should be modelled through a small set of independent sub-models.

The referred roots allow us to propose a design methodology described by Figure 1.

The system' initial requirements are kept using UML use cases [2]. In this sense, capture of complex and also primitive functionalities are obtained in an informal way, constructing a set of use cases, which can be afterwards validated by users at the very beginning of the design process. In this process, different levels of perception of the system's functionalities are possible, using hierarchical structuring mechanisms common in UML techniques. The different actors that interact with the system are identified, and their interrelations as well.

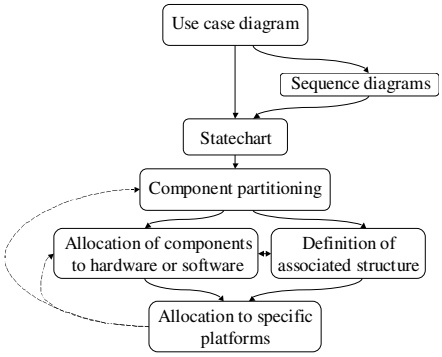

Figure 1. Methodology overview.

The next step is to produce a model of the system that can be used both for specification and for implementation. From the goals characterised in the introductory section, statecharts [3] [4] were selected to be the specification formalism to accommodate behavioural modelling. We can benefit from the fact that the same model that we use to produce a specification model can be also used as the basis for the implementation.

In this sense, we need a way to produce the system model starting from the use cases description. Two ways are foreseen:

- Directly translate each use case into a statechart (or a state diagram);
- Translate each use case into a sequence diagram, which is in turn amenable to be represented by a statechart [5].

From our experience till the moment, we only used direct translation. However, for some applications, it is understood that sequence diagram usage seems to be very valuable. Procedures enabling the translation of sequence diagrams into statecharts are known from the literature and can be applied for this task.

The system model will be built based on the parallel composition of the sub-models associated with each use case.

The next (obvious) step is to face the partitioning of the system into several components. Each component will be afterwards mapped to software or hardware, according to the price and performance that one wants to reach.

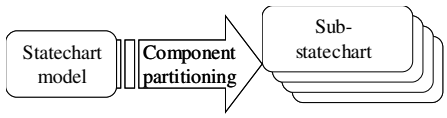

Figure 2. Partitioning of the model.

In this sense, from the initial statechart, we will obtain a set of statecharts that will be the result of the decomposition of the initial model. This set of statecharts will be executed concurrently.

The set of criteria applied to produce the desired partitioning may use, at some extend, ad-hoc techniques based on "engineering common sense practice" attitude. At the end of the process, the solution can be evaluated in terms of price and performance and different partitions can be tried. Analysis of specific metrics to produce the model partition is out of the scope of this paper.

Each one of the sub-models produced will be allocated to a specific platform and the resulting implementation platform structure is illustrated in Figure 3.

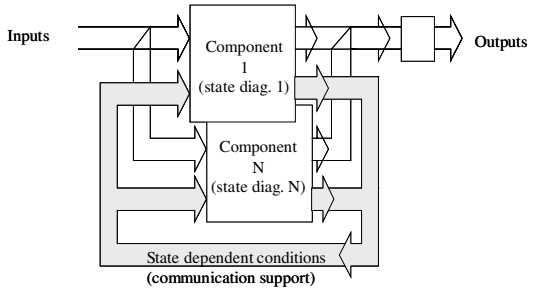

Figure 3. Implementation structure overview.

We would like to emphasise that as a consequence of the model partitioning into different components, a new concern is created: communication resources among the different components. In this paper no special reference is explicitly produced regarding these issues. So, from the point of view of the presented works, the secure communication between the different components is always assured, through the explicitly modelling in the initial model of the handshake between the different components, either using a global synchronising signal (global clock) or as in the so-called GALS systems (Globally Asynchronous Locally Synchronous systems).

Proceedings of the First ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE'03) ISBN 0-7695-1923-7/03 $17.00 © 2003 IEEE

3. Statecharts

Statecharts [3] [4] have been used for system modelling and also as implementation specification; different tools are available for their use, namely Statemate [1] and Rhapsody [6].

Statecharts are a graphical formalism based on state diagrams, plus the notions of hierarchy, parallelism and communication between parallel subsystems. Those characteristics were key points that supported its adoption as one of the specification formalisms within the UML community [7].

Statecharts defines two types of refinements, namely the XOR refinement and the AND refinement. The XOR refinement supports the hierarchy concept, through encapsulation of state machines. The AND refinement supports the concurrency concept, through parallel execution of XOR components. In this sense, it is common to consider three types of state instances: the set (associated with the AND refinement), the cluster (associated with the XOR refinement) and the simple state (named in this way in [8]).

The default state, associated with every state diagram presented in the model, defines which state will take control when the associated state machine is activated, (for instance when a transition reaches a cluster state). Figure 4 shows the basic notation to represent a state diagram.

Transition expressions, associated with arcs between states, are composed by events and conditions (see Figure 4). Internal or output events can be generated as a consequence of the firing of the transition (Meally machine). Also, output actions can be associated with states (Moore machine).

Transition expressions can use special events, special conditions and special actions. Examples of special events are the *entered(state)* event and *exited(state)* event, which are generated whenever the system respectively enters or exits some state. Example of special conditions is *in(state)*, which indicates that the system is currently in some state. Examples of special actions are *clear history(state)* and *deep clear history(state)*, which initialises the history state of the cluster named *state* to the default state.

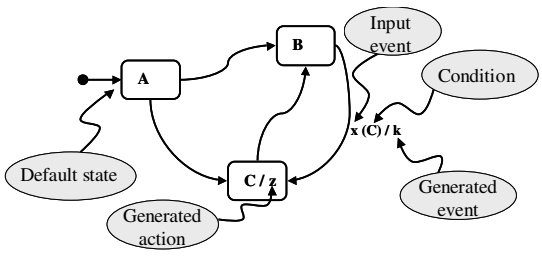

Figure 4. Statechart basic notation.

Apart from the referred main characteristics, the statecharts formalism presents some convenient features, namely the concept of history. The history concept supports, among other things, the modelling of interrupts, allowing automatic context restoring after the completion of the interruption service procedures. The history concept can be associated to cluster state instances. When the system enters a cluster with history attribute, the state that will be active upon entrance will be the one that was active upon the last exit in time from that cluster. In the case of the first entrance in the cluster, the active state will then be the default one. The history attribute can be simple (in the way it was presented) or deep; deep-history means that all the clusters inside the cluster that has that property also have that property.

So far we only characterised syntactic issues for the statecharts formalism. Its semantic characterisation includes, among other things, the characterisation of the step algorithm and the definition of the criteria for solving conflicts (namely choosing a set of triggering transitions among a set of conflicting transitions) [1] [9].

The step algorithm imposes the way in which the system evolves between two states and is dependent on the chose semantics. In [1], the simultaneous trigger of a finite set of transitions is called micro-step. Transitions fired in a micro-step can generate internal events that, in turn, can fire other transitions. Thus, the definition of step is a finite set of micro-steps, which means that after all the micro-steps take place, there will be no more active transitions and the system will remain in a stable state. External events are not considered during a step, so it is supposed that a system can completely react to an external event before the occurrence of the next external event. In this sense, the broadcast of events occurs in zero-delay time. Different semantics associated with the handling of broadcasted event are discussed in [9].

4. Running example – presentation

The present section briefly describes part of an application that used the presented methodology for the implementation of an embedded system targeted for image acquisition, storage and visualization, including a user interface for navigation on the acquired image. In a broad sense, it is a monitoring system composed by a set of cameras and a user interface that enables navigation through the set of images acquired from the set of cameras (where an output image can be composed by parts of images obtained from different cameras). The emphasis was put on the hardware-software co-design of the system.

Although, according to the specific very tight time constraints of the system, the focus of most of the component implementation is on the hardware part. The software part of the system was chosen to be associated

Proceedings of the First ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE'03) ISBN 0-7695-1923-7/03 $17.00 © 2003 IEEE

mostly with the user-interface, which is not hard real-time constrained. For future extensions of the system, namely implementation of remote communications, emphasis on software-based components is foreseen.

The modelling process starts with the identification of the relevant use cases. Figure 5 presents a diagram of the produced use cases.

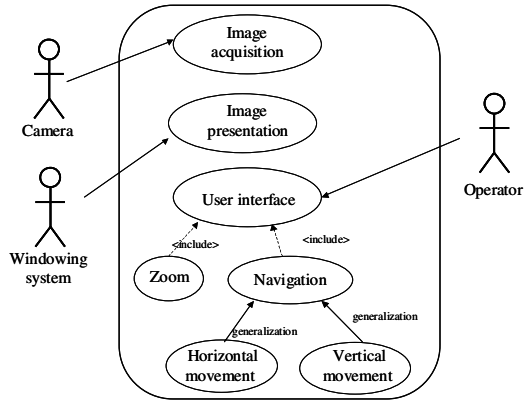

Figure 5. Use cases diagram.

For every use case identified, a specific statechart was built. The top-level statechart model is presented in Figure 6, where the three main activities are associated with the three main use cases: image acquisition, image visualization and user interface.

Figure 7 presents a refinement of the top-level model and Figure 8 a further refinement. More specifically, Figure 7 presents the refinement associated with the set Acquisition of Figure 6, and Figure 8 presents the refinement of the Pixel_counter state of Figure 7.

In both figures, a parameterised representation of a statechart is used (which is an extension to the syntax used in common statechart development commercial environments), as far as it enables a very compact representation of the specific functionality to be modelled.

Most of the states at the second level models need to be refined into a third level model. In this way, we can conclude that the referred system needs to be modelled using several levels of decomposition (we may eventually conclude that it is a medium complexity system).

5. Partitioning into components

At this point, we come to the next step: the partitioning of the model.

The goal is to start with a statechart model and end-up with a set of concurrent statecharts, obtained from the partitioning of the initial one.

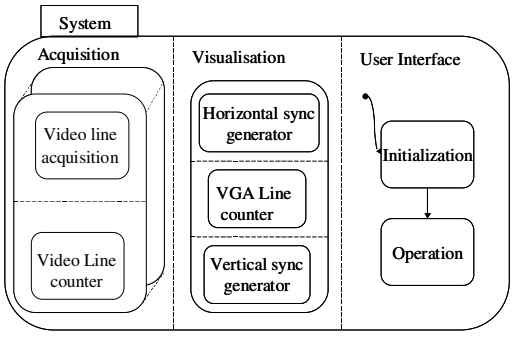

Figure 6. High-level statechart model.

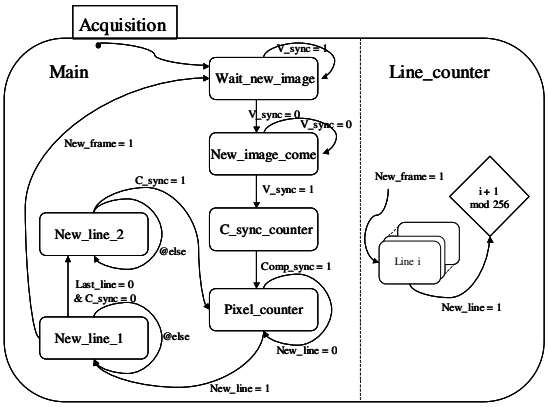

Figure 7. Second level model.

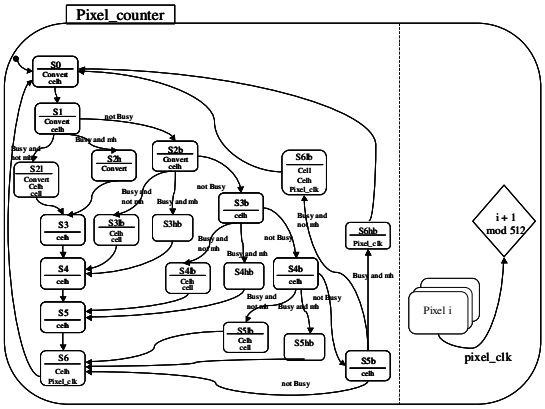

Figure 8. Third level model.

In this sense, we can find two situations:

- Decompose a state diagram into several concurrent state diagrams ("state sub-diagrams");
- Decompose a statechart into several concurrent statecharts ("sub-statecharts").

Next subsections address these two goals.

5.1. State diagram partitioning

The partitioning of a state diagram into N concurrent communicating state diagrams, obtained from the first one, is a simple task (as far as we consider that the communication between the obtained components is secure, it was solved long time ago, and will be revisited here through a new perspective). From a formal point of view, we want to obtain a result that is equivalent to a statechart (with only one AND set). Of course, that is assured that the behaviourally equivalent state space associated with the obtained statechart is isomorphic with the initial state diagram. Figure 9 illustrates the process.

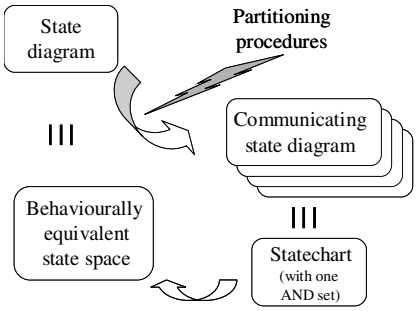

Figure 9. State diagram partition.

Considering that we can identify strong connected components in the original state diagram, and use them as our criteria to split the model, we may consider two typical situations:

- Existence of two strong connected sub-diagrams, interconnected by one arc (elementary case);
- Existence of N strong connected sub-diagrams, interconnected by an arbitrary number of arcs between them (general case).

Even we are not able to identify strong connected components, and we produce a partition based on other criteria, the translation procedures presented will be applicable, as far as the cutting transition set is defined.

Let's start by the elementary case, where the goal is to obtain a statechart with two concurrent communicating state diagrams. The procedures to apply are the following:

- Each of the decomposed state diagrams is composed by the set of states and arcs of each strong connected component;
- In the state diagram associated with the first strong connected component (the one which has the default state), one new state is added representing the second strong connected component (from the point of view of the first strong connected component, the whole second strong connected component collapses into this new state);
- In the state diagram associated with the second strong connected component, one new state is added representing the first strong connected component; this new state is the default state of the second state diagram (from the point of view of the second strong connected component, the whole first strong connected component collapses into this new state); the condition associated with the arc connecting the new state and the entering state of the second strong connected component will include a condition on the activation of the origin state.

Figure 10 illustrates the procedure.

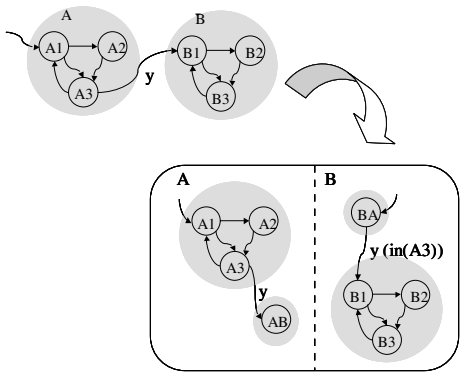

Figure 10. State diag. partition: elementary case.

Coming to the general case, consider Figure 11 where three strong connected components is presented, containing arcs interconnecting any one of the components to any other. The procedures to apply result from the observation of the elementary case, just presented, and can be summarised by the following:

- Each of the new concurrent state diagrams is composed by the set of states and arcs of each strong connected component;

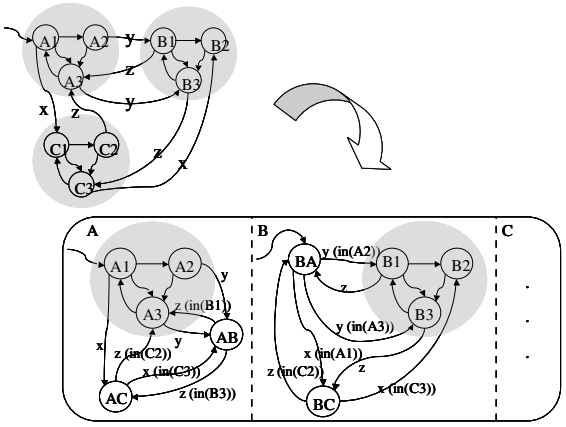

Figure 11. State diagram partition: general case.

Proceedings of the First ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE'03) ISBN 0-7695-1923-7/03 $17.00 © 2003 IEEE

- For each new state diagram, a new state is added for every component connected to the associated component; arcs are created accordingly to the initial model;
- Default state for each component is elected according to the default state of the initial model;
- The condition associated with every arc belonging to the cut set will include a condition on the activation of the origin state.

Figure 11 shows an example of application of the referred procedures.

5.2. Statechart partitioning

Considering that the system is modelled by a statechart, the goal of the partitioning procedure is to obtain a set of concurrent communicating state diagrams, as obtained in the previous subsection. Figure 12 illustrates the goal.

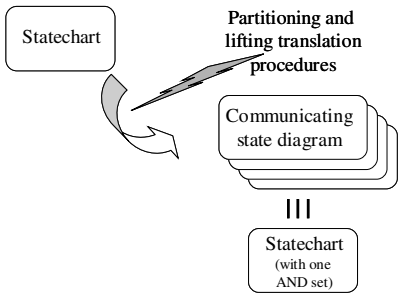

Figure 12. Statechart partition.

We can consider two situations:

- When the cut is coincident with the parallel components of the statechart (and the problem is solved, directly or using the procedures presented in the next section);
- When the cut needs to consider the splitting of some state diagram component, and the procedures of the previous section applies.

6. Implementation of statecharts

Several references can be found in the literature, addressing statechart implementation. However, most of them only address software-based implementations.

Figure 13 presents a roadmap for possible design flows based on statechart modelling [10]. While specification, simulation and verification of proprieties tasks are carried on based on the statechart diagram and on the associated state space, the tasks associated with the implementation can be spited into two main groups:

- Direct implementations: based on translation of the statechart components;
- Indirect implementations: based on a previous "translation" of the statechart into the associated state space.

The second strategy is particularly interesting whenever one wants to implement a statechart specification into a very low-cost platform (low-cost micro-controller, for instance); in this situation one does not have the necessary resources to support non-trivial statechart characteristics implementation (like broadcast communication and multilevel clustering); also, most of the time in these cases, we do not want to implement a complex design, but only a small to medium complexity system. In this sense, the implementation specification is based on the flat model that is behaviourally equivalent to the original statechart. For those low-cost platforms cases, the state space based implementation proved to be effective [11].

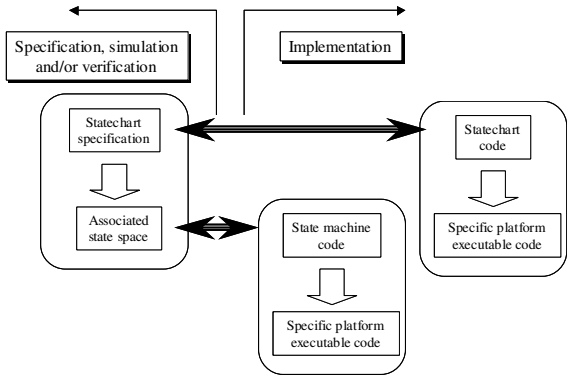

Figure 13. Statechart implementation strategies.

The first strategy (direct implementation) is the most common, by far. It is the case for two of the most well-known development environment that use statecharts, namely Statemate [1] and Rhapsody [6].

One can identify three main strategies for direct implementations, targeted for software-based implementations [5]:

- via switch-statements [2]; applicable only for simple cases;
- via State Pattern [6]; classical object-oriented replacement of switch-statements (one class for any state), and statechart structure becomes class hierarchy (static code);
- via State Table [7] [12] [13]; statechart structure becomes runtime object structure.

In the present work, we propose to, in some sense, stay at the middle of the two strategies, as far as we do not want to pay the price for a flat specification loosing concurrency (as for state space based), but we also want to address hardware implementation (and also low-cost platforms).

The proposed "balanced" attitude is targeted to accomplish statechart execution through a direct implementation strategy (to avoid flattening the model), but, at the same time, do not rely on complex features like zero-time delay communication and hierarchical refinements. This goal is accomplished using several translation procedures of the initial statechart model allowing the lifting of the statechart structuring characteristics, in order to obtain a top-level model composed by a set of concurrent components

Using the already presented techniques for partitioning statechart models, we can identify different components, each one amenable to be implemented in hardware or software.

We argue that in this way we have a coherent usage of statecharts in a co-design framework, enabling formal verification prior to implementation.

The different characteristics translation procedures will be presented individually in the following sections, in a informal way. In [14] most of them were introduced.

6.1. Depth

Let's start with XOR refinement. From the modelling point of view, it supports the capability to interrupt an activity modelled through the usage of a cluster.

Consider the case of a two-level model (the simplest one). At the upper-level we have simple states and one cluster containing a dependent state diagram at the lower level (refer to Figure 14). The proposed translation procedures take advantage on the fact that we can have concurrent activity at both levels; although, activity at the upper-level will deactivate the cluster state activity and the dependent state diagram will be non active.

The translation procedure to be applied to a cluster is the following (if we consider a state diagram with N clusters, the referred procedure can be applied considering N+1 resulting components):

- The model will be translated to an AND set at the top-level, composed by two components
- The first component is composed by the initial upper-level state machine (original simple states and a simple state representing the original cluster and by all the associated transitions);
- The second component is the result of the lifting of the contents of the cluster state to the upper-level, and is composed by a state diagram generated as follows:
  - A simple state (A0, for instance), representing "*Cluster A no active*" is added to the state machine and considered as the default state;
  - All the simple states of the dependent state diagram are kept;
  - A transition is added from the new default state to the original default state of the cluster A; the expression *entering (A)* is associated with that transition;
  - Transitions are created from every state in the cluster to the "*Cluster A no active*" state; expression *leaving (A)* will be associated with those transitions; those transitions are considered to have higher priority regarding the original transitions already presented in the model (in order to avoid non-determinism).

One simple situation of lifting one XOR cluster is presented in Figure 14.

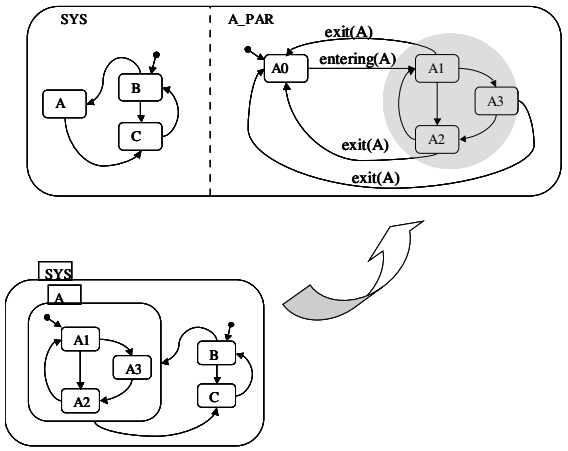

Figure 14. Lifting depth concept.

As far as the associated state spaces can prove to be isomorphic, the resulting model is behaviourally equivalent to the initial one.

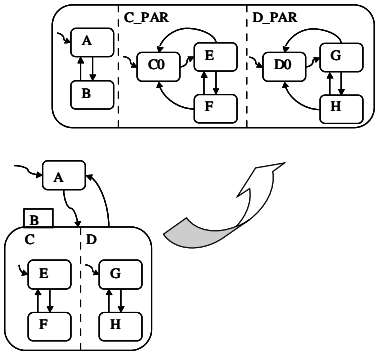

Figure 15. Lifting orthogonality concept.

6.2. Orthogonality

Let's come to AND refinement. In this case, the translation procedures proposed for lifting of the XOR cluster will be applied for every parallel component of the AND cluster. Figure 15 illustrates a simple example.

6.3. Multilevel models

The simultaneously usage of AND and XOR refinements does not put new situations, as far as the referred procedures still be valid and should be applied iteratively starting from the inner levels, lifting level by level till reaching the top-level with a set of parallel components grouped in a AND set.

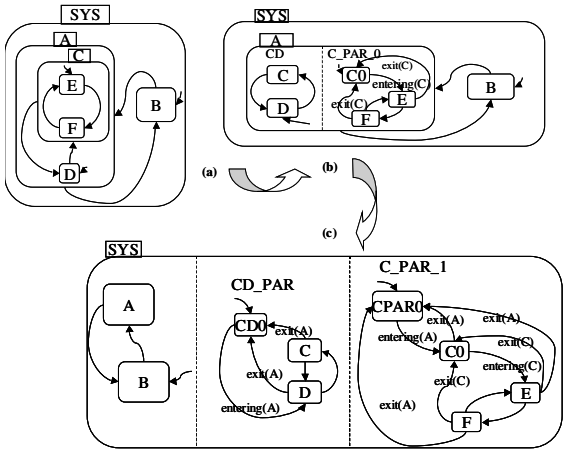

Figure 16. Lifting of inner levels.

Figure 16 shows an example:

- We should start by the lifting of the contents of cluster C, {E, F}; in this sense, cluster A contents will be translated into a set of two parallel components (composed by states {C, D} and {C0, E, F}, accordingly);
- After that, we will take care of the lifting of the set A, obtaining a set composed by three parallel components, {A, B}, {CD0, C, D} and {CPAR0, C0, E, F}.

6.4. History concept

Other statechart feature, the history concept can be associated with every cluster state. Its usage allows the definition of complex procedures like interruption, enabling automatic context restoring after completion of the interrupt service procedure.

The associated translation procedures add the following to the already proposed procedures for lifting a cluster:

- The "Cluster no active" added state has the meaning of "Cluster no active and never activated before";
- For every state *Si* of the dependent state machine, a simple state *Si'* is added to the state machine, as a phantom of *Si*, representing "*Cluster no active and the last active state was Si*";
- A transition, with the associated expression *entering(Cluster),* is added from *Si'* to *Si*; another transition is added from *Si* to *Si',* with the expression *leaving(Cluster)* attached; in this way, the set of phantom states will be used to store the last active state within the cluster;
- As an optimizing step, the phantom state of the initial state can be suppressed (merged with "Cluster no active and never activated before").

Figure 17 illustrates the procedure.

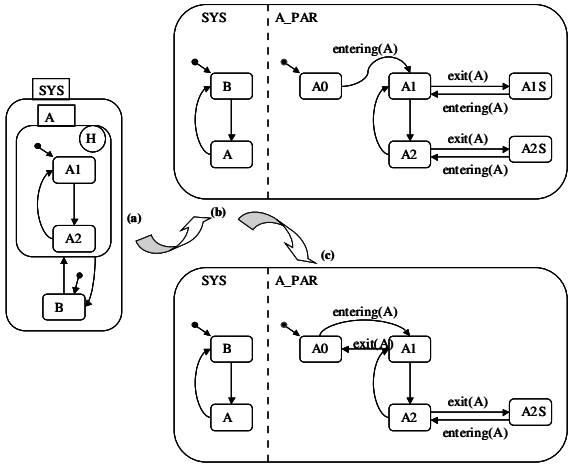

Figure 17. History concept.

For deep-history concept translation, similar procedures are applicable, considering its specific semantics (applied for every cluster contained in the cluster receiving the deep-history attribute).

6.5. Communication support

From the implementation point of view, the zero-delay time paradigm is the most challenging and "dangerous" concept proposed within statecharts.

Several semantics have been proposed for its implementation [1] [9]. As far as we intend to use statechart models for the specification of systems independently of the type of implementation platforms to be used, namely software-only, hardware-only and including hardware-software partitioning (through co-design techniques), issues associated with event broadcast have to be adequately prepared.

Considering that the evolution of the model will be accomplished only at specific instants in time, it is necessary to assure that all the evolutions associated with the broadcast events to be generated during the associated micro-steps will be accomplished using a "look-ahead" technique, in order to foreseen its occurrence. We have to consider two types of situations:

- When the generated event and further generated events (associated with the consequences of its occurrence) only affects new parallel components;
- When generated events will affect the evolution of state diagrams already changed in the current step of analysis.

The last situation will not be further commented, as far as it is considered as a "bad modelling practice" [14].

So, we only address the first situation and all the references to a generated event (which will be received through the broadcast mechanism) should be replaced by an equivalent expression that can be completely evaluated at the beginning of the step (which means the clock period for hardware implementations or the execution cycle for software implementations).

This expression will include references to the external event that causes the beginning of the step and to conditions reflecting the internal structure of the model that can be evaluated at the beginning of the step (implementing the desired look-ahead behaviour).

Figure 18 illustrates the procedure for simple situations.

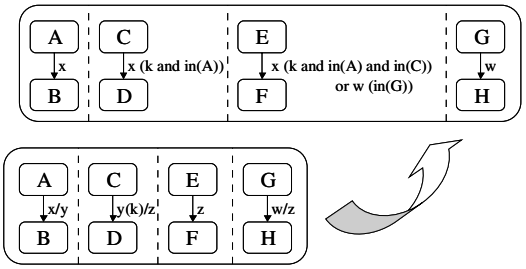

Figure 18. Event look-ahead generation.

As far, the proposed set of translation procedures for the different statecharts characteristics relies on the usage of the internal events *entering(state)* and *leaving(state)*.

These events can be defined based on the conditions of activation and deactivation of the state.

Regarding the *entering(state)* event, it can be computed considering the incoming arcs that reach *state*, which means

$$entering(state) = \Sigma (\text{all event/conditions that causes activation of state})$$

In a similar way, the *leaving(state)* event can be determined considering the outgoing arcs leaving *state*, namely

$$leaving(state) = \Sigma (\text{all event/conditions that causes deactivation of state})$$

For Figure 19, the events associated with state A are:

entering(A) = v (in(B)) or z (p and in(E)

leaving(A)= (x or y (K)) and in(A)

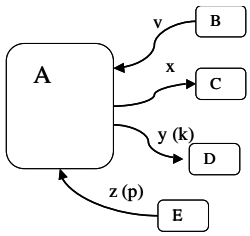

Figure 19. Special events.

6.6. Non-well-structured statecharts

Finally, as far as the designer has plenty of freedom to produce non-structured models, we should consider the existence of "non-well-structured" transitions to or from the cluster or set states.

For these situations, additional steps have to be included to the proposed procedures.

Let's start with the existence of incoming arcs that will end in internal states of the cluster (and not at the border, as in a structured model). The proposed procedures will be changed in the following way:

- For every incoming arc at the cluster, a transition will be created from the "*Cluster no active*" state to the corresponding destination state (initial procedures foreseen to use default state for arcs ending at the border of the cluster);
- The expression associated with a new transition is composed by *entering(A)* (as in the initial procedure), anded with the original associated expression.

Figure 20 illustrates one example.

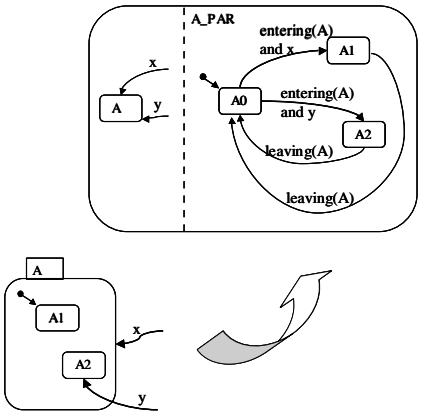

Figure 20. Lifting of multi-incoming (activating) arcs.

The second case is associated with outgoing arcs, leaving from internal states.

Proceedings of the First ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE'03) ISBN 0-7695-1923-7/03 $17.00 © 2003 IEEE

In this case, the procedure includes the lifting to the border of the cluster of every outgoing arcs that starts at an internal state; the transition expression associated with every arc will include the condition of being in the starting state, through the inclusion of *in(starting_state)* to the expression. Figure 21 illustrates the situation.

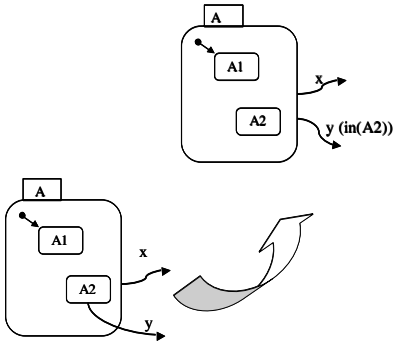

Figure 21. Lifting of multi-outgoing (deactivating) arcs.

The steps presented in the present section have to be performed before the previously mentioned procedures of lifting of the cluster contents.

8. Running example - closure

The referred procedures were successfully applied to the described monitoring system. From the point of view of the implementation platform, it contains programmable logic devices (one 95144 CPLD per camera and one Spartan-II FPGA from Xilinx), and a high-performance dual-port RAM per camera (apart from other hardware, specific for video acquisition and generation). VHDL was used as the implementation language. Some parts of the user interface were implemented using a micro-controller IP from Xilinx (Picoblaze) [15].

9. Conclusions

The proposed methodology starts from grabbing user requirement based on use cases and translate them to statecharts allowing the embedded system designer to use formal techniques for propriety verification.

Afterwards a set of procedures for the translation of statecharts into a behaviourally equivalent statechart was presented. The resulted statechart is amenable to be implemented either in software or in hardware. In this sense, the usage of co-design techniques is fully supported.

The methodology proved to be effective in the design of a monitoring system.

References

[1] David Harel, Michal Politi; "Modeling Reactive Systems with Statecharts – The STATEMATE Approach"; McGraw-Hill; 1998; ISBN 0-07-026205-5

[2] Grady Booch, James Rumbaugh, Ivar Jacobson; "The Unified Modeling Language User Guide"; Object Technology Series, Addison-Wesley; ISBN 0-201-57168-4, 1999

[3] David Harel, "On Visual Formalisms", Communications of the ACM, vol. 31, number 5, May 1988, pp. 514-530.

[4] David Harel, "Statecharts: A Visual Formalism for Complex Systems", Science of computer Programming, vol. 8, 1987, pp. 231-274.

[5] Albert Zundorf, "Code Generation from UML Behavioral Diagrams", Tutorial F, Fifth International Conference on the Unified Modeling Language, Sep.30 – Oct 4, 2002, Dresden, Germany

[6] Rhapsody; "The Rhapsody case tool reference manual; Ilogix, http://www.ilogix.com/

[7] Bruce Powel Douglass, "Real-time UML – developing efficient objects for embedded systems", Object Technology Series, Addison-Wesley, 1998, ISBN 0-201-32579-9

[8] Paul Jay Lucas; "An object-oriented language system for implementing concurrent, hierarchical, finite state machines"; MSc Thesis; Graduate College of the University of Illinois at Urbana-Champaign; 1993.

[9] Michael von der Beeck, "A Comparison of Statechart Variants", in Formal Techniques in Real-Time and Fault-Tolerant Systems, Lecture Notes in Computer Science, vol. 863, Springer-Verlag, 1994, pp. 128-148.

[10] Luís Gomes, Carlos Soares; "Low-cost embedded systems design using Statecharts"; 2001; 5th WSES/IEEE World Multiconference on Circuits, Systems, Communications & Computers, CSCC'2001; 8-15 July 2001; Crete, Greece

[11] Carlos Soares, "Utilização de Statecharts em Controladores Programáveis - da especificação à implementação" (Statecharts usage for programmable controllers – from specification to implementation) (in Portuguese), MSc Thesis, Univ. Nova Lisboa, 1998.

[12] Bruce Powel Douglass, "Doing Hard Time – Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns", Object Technology Series, Addison-Wesley, 1999, ISBN 0-201-49837-5

[13] H. Köhler, U. Nickel, J. Niere, A. Zündorf; "Integrating UML Diagrams for Production Control Systems"; Proceedings of ICSE'2000 – The 22nd International Conference on Software Engineering, June 4-11, Limerick, Ireland, ACM Press; 2000

[14] Luís Gomes, Anikó Costa; "On Lifting of Statechart Structuring Mechanisms"; Proceedings of ACSD'2003 Third International Conference on Application of Concurrency to System Design ; 18-20 June 2003; Guimarães, Portugal

[15] Ken Chapman; "Picoblaze"; www.xilinx.com