A Case Study on Controller Synthesis for Data-Intensive Embedded Systems

Abdoulaye Gamatié, Huafeng Yu
LIFL/CNRS - INRIA Lille Nord Europe
Lille, France
Email: {Abdoulaye.Gamatie, Huafeng.Yu}@inria.fr

Gwenaël Delaval, Éric Rutten
INRIA Rhône-Alpes
Grenoble, France
Email: {Gwenael.Delaval, Eric.Rutten}@inria.fr

Abstract

This paper presents an approach for the safe design of data-intensive embedded systems. A multimedia application module of last generation cellular phones is considered as a case study. The OMG standard profile MARTE *is used to adequately model the application. The resulting model is then transformed into a synchronous program from which a controller is synthesized by using a formal technique, in order to enforce the safe behavior of the modeled application while meeting quality of service requirements. The whole study is carried out in a design framework,* GASPARD, *dedicated to high-performance embedded systems.*

1. Motivation

1.1. Data intensive computation and control.

Modern embedded systems increasingly adopt high-performance execution platforms. Concerned applications are, e.g., state-of-the-art multimedia systems such as high-definition digital television (HDTV), biometric data processing, sonar and radar signal processing. In all these domains, a common feature is that the systems perform *data-intensive* algorithms. E.g., in a HDTV, such algorithms include downscaling or upscaling, which enable to make the displayed images smaller or bigger respectively. These operations are applied to millions of pixels, representing a large amount of data. In order to conveniently achieve their function, such high-performance systems have resource constraints: memory capacity, processor load, energy, etc. For an efficient resource management, some adaptivity and reconfigurability mechanisms are needed so as to enable a flexible system execution w.r.t. environment and platform resource constraints. These control mechanisms can be applied to, e.g., power-aware management, fault-tolerance and recovery. The above issues lead to a challenging question about the definition of reliable design approaches for embedded systems, mixing data-intensive computations and control.

1.2. A safe design approach.

We propose an approach based on a *model-driven engineering* (MDE) framework, called GASPARD [1], [2], in which systems are specified with the OMG standard UML profile dedicated to *Modeling and Analysis of Real-time and Embedded systems* (MARTE) [3]. Model-based approaches carry out the system design at a high level of abstraction, and the back-end model transformations enable to automatically generate implementation code. The modeling concepts provided in GASPARD offer an efficient way to specify the parallelism that is inherent to both functional (data-intensive algorithms) and non functional (execution platform) parts of high-performance embedded systems.

In this paper, we address safe design *particularly for the control part*, thereby separating concerns from others, more data-related transformation and validation, which are also necessary and treated elsewhere [4], [1], [2]. We perform MDE transformations towards synchronous models, which are *very close to the source level*, and just constitute a verifiable representation. We particularly address the synthesis of a safe application controller enforcing correct behaviors w.r.t. functional and non functional requirements. The usage of formal methods with a user-friendly modeling language support such as GASPARD, is very helpful in order to accelerate the validation process while reducing the system design cost.

The *logical relationship* between these different aspects is summarized by the following roadmap: the source level specification in GASPARD (Section 2) contains control-related parts; these are extracted and transformed to an intermediate synchronous verifiable automaton model (Section 3), which is used to automatically synthesize a complete controller for given properties (Section 4); the controller is integrated into the automaton model; from then on, GASPARD features further transformations and compilation techniques to proceed towards machine code.

1.3. Case study: a multimedia application.

Following the recent technical advances, multimedia mobile devices are spreading rapidly. Typical examples are modern cellular phones, which have complex functionalities: camera, games, mp3 music, video. Let us focus on the video part. The played clips are obtained either on-line or from the local memory. There are different display modes such as *Black & White*; *Negative*, a tonal inversion style of a positive

978-0-7695-3678-1/09 $25.00 © 2009 IEEE
DOI 10.1109/ICESS.2009.12

image; *Sepia*, a dark brown-grey color style; or *Normal*, meaning no effect. In addition, the resolution of the video can be set to *High*, *Medium* and *Low*. Finally, the color can be in either *Color* or *Monochrome* options. Besides user commands, the video display modes are controlled by the system according to quality of service (QoS) requirements, including the status of computing resources, the energy level, the communication quality.

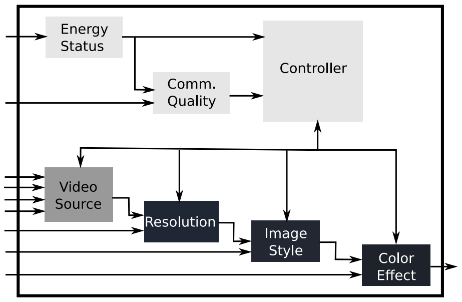


Figure 1. Multimedia functionality module.

Our case study is a multimedia processing module (see Fig. 1) composed of the following parts:

- *EnergyStatus*: indicates the energy level according to events received from an energy monitor component;
- *CommQuality*: provides the communication quality level, according to the energy level and the on-line transmission bandwidth of received data;
- *Controller*: validates mode change requests from other components according to current mode configuration and the available computational resources. This specific component will be automatically obtained by *discrete controller synthesis*;
- *VideoSource*: chooses an appropriate video source according to user requests and *Controller*'s permission;
- *Resolution*: computes an appropriate resolution according to user requests and *Controller*'s acceptance. It includes a data-intensive processing part;
- *ImageStyle* and *ColorEffect*: similar to *Resolution*, but compute the appropriate image style and video color effect respectively.

The above module has different configuration modes following which its components achieve algorithms for a suitable image display. Depending on the resource status, e.g., energy level, communication bandwidth, the display quality of an on-line video varies. Hence, each mode is associated with non-functional properties, which must be satisfied in order to display images at a good quality level. This paper proposes an approach that deals with these aspects so as to safely guarantee that the designed multimedia functionality module actually meets its specification requirements.

2. The GASPARD design framework

In the GASPARD framework, a system-on-chip (SoC) under design is described with the MARTE profile (Fig. 2). This profile extends UML with concepts that can be used to model the software and hardware parts as well as their association and platform deployment. Such models contain the useful information that enable to address different design issues: parallelism, scheduling, performance evaluation, etc. More precisely, the SoC design in GASPARD relies on the *repetitive* model of computation (MoC) [4] (called *repetitive structure modeling* - RSM in MARTE). This MoC is inspired by the industrial domain-specific language Array-OL [5] from Thomson Marconi Sonar. It manipulates multidimensional structures, e.g., arrays, and offers a factorized way to describe both *task parallelism* and *data parallelism*. It is efficiently compiled towards high-performance architectures such as single instruction multiple data (SIMD).

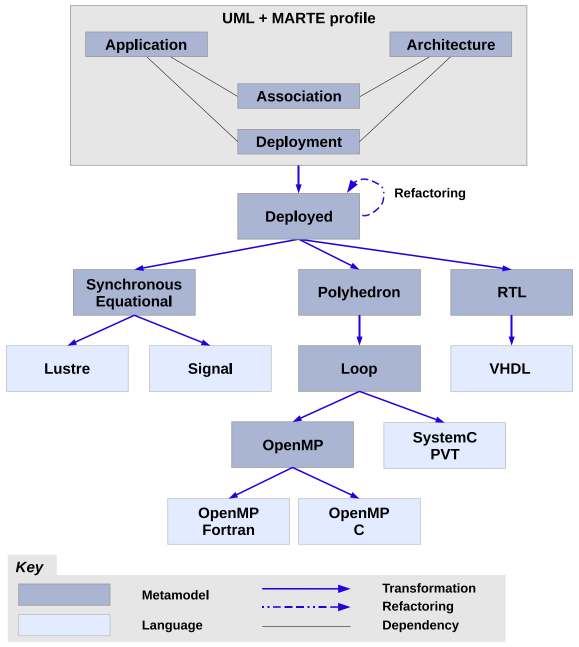


Figure 2. The GASPARD design framework.

GASPARD adopts a component-based approach. An application is specified as a component dependency graph (see a concrete example in Fig. 7). The components are connected via ports stereotyped as *FlowPort*. Each port has a shape (or dimension). The elementary components are implemented by Intellectual Properties (IP) enriched with useful information for performance analysis. This corresponds to the *Deployment* phase shown in Fig. 2.

The high-level models are refined towards specific technologies via automatic model transformations: synchronous languages (Lustre, Signal) [6] for formal validation, SystemC for simulation, OpenMP Fortran and C for execution and VHDL for hardware synthesis. According to MDE principles, at each step of the refinement, the concepts are characterized by a dedicated metamodel. The backbone environment that implements this methodology is Eclipse.

2.1. Repetitive structure modeling

An example specified with RSM is shown in Fig. 3. It expresses data-parallelism in a monochrome filter, *MonoFilter*, used for the processing of a [320, 240]-image. Because it only works on small [8, 8]-pixel subsets, it should be repeated 40×30 times to cover a whole image. In RSM [40, 30] is referred to as the shape of *repetition space* associated with *MonoFilter*.

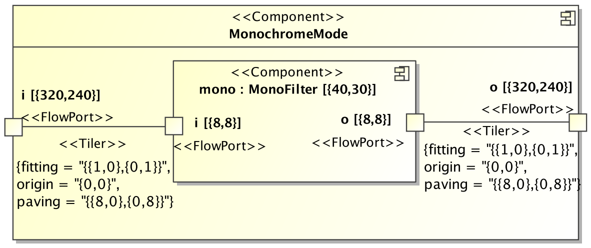


Figure 3. A monochrome effect filter.

The repeated *MonoFilter* component runs in a repetition context, defined by the *MonochromeMode* component. All repetition instances run in parallel. A *Connector* used in a repetition context is called *LinkTopology*. It adds a set of topological information to a UML *Connector*. *MonoFilter* is connected to *MonochromeMode* via *Tiler* links. The repetitions of *MonoFilter* consume and produce identically shaped sub-arrays of pixels, which are respectively extracted from the input port *i* and stored in the output port *o*. These sub-arrays, referred to as *patterns* (shaped [8, 8] in the example), are tiled according to the *Tilers* by using the following information [4]: i) an *origin* of the *reference pattern*, ii) a *paving* matrix specifying how patterns cover arrays, and iii) a *fitting* matrix specifying how array elements fill each pattern.

2.2. Specification of controlled computations

GASPARD adopts the state-based control, which is inspired from the reactive mode automata [7], [8]. A task under control may have several exclusive running modes. The mode activation at run-time is determined by automata. Two kinds of components are distinguished: *mode switch component* (MSC) and *state machine components* (SMC).

An MSC achieves a switch function between different modes as shown in Fig. 4 where the *ColorStyleSwitch* component has two modes *ColorMode* and *MonochromeMode*. It has *mode_color* and i as inputs and o as outputs. *mode_color* is a UML *behavior port*, which conveys mode

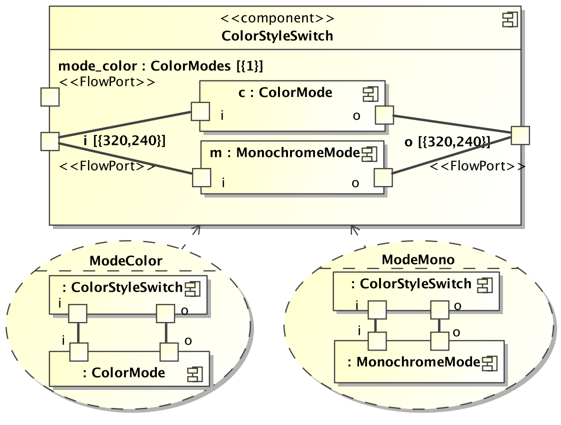
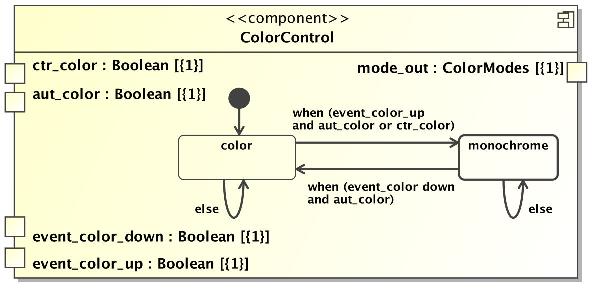


Figure 4. Switch of color effects.

values. According to these values, modes are activated at each instant. The activation behavior is specified by UML collaborations. For instance, the *ColorMode* component is executed only in the mode *ModeColor*, which is indicated by the name of the collaboration.



<<component>>
ColorControl
ctr_color : Boolean [{1}]
aut_color : Boolean [{1}]
mode_out : ColorModes [{1}]
when (event_color_up
and aut_color or ctr_color)
color
monochrome
when (event_color down
and aut_color)
else
else
event_color_down : Boolean [{1}]
event_color_up : Boolean [{1}]

Figure 5. A color effect controller.

An SMC determines the mode values used by MSCs to execute different computation modes. This component is associated with UML state machines. Each state is associated with a mode. As shown in Fig. 5, the *ColorControl* SMC has an interface that includes the input Boolean ports *ctr_color*, *aut_color*, *event_color_up* and *event_color_down* and output mode port *mode_out*. The values from input ports are dispatched to trigger transitions. The transition conditions are prefixed by *when*. A mode value specified in a state, is conveyed through the *mode_out* port. An important required property of the state machines is determinism.

Mode automata [7] can be constructed from the composition of an SMC and an MSC as illustrated by the *ColorEffect* component in Fig. 6. This typical composition should be placed in a repetition context with *inter-repetition dependencies* (IRD) [9]. The reasons are twofold: *ColorEffect* suggests the processing of one frame of a video clip, so it should be repeated; an IRD specifies the sequential processing of these frames. Parallel mode automata and hierarchical mode automata are specified in a similar manner: automata defined

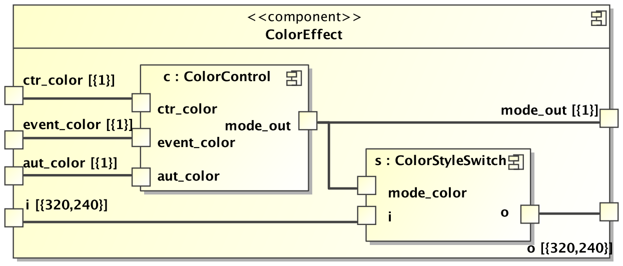


Figure 6. A composition of SMC and MSC.

in the same repetition context can be composed in parallel, because they have the same reaction pace; the automata specified in a mode are considered as sub-automata of the automaton that controls this mode.

2.3. Global model of the multimedia module

A global model view of the multimedia module is shown in Fig. 7. The component `CellPhoneExample` processes only one video frame. So, it must be repeated to process a video clip. The ports on the left-hand side of the component represent its inputs, mainly events and image sources. The only output is the processed image. The `CellPhoneExample` model contains instances of the components introduced in Section 1. In addition to data-intensive computations, the components `VideoSource`, `ColorEffect`, `ImageStyle` and `Resolution` contain state machines, illustrated in Fig. 9. These state machines define the different modes that form a global configuration of the multimedia functionality module. The controllability of these state machines is modeled by the presence of two types of controllable labels: those prefixed by "`aut_`" represent the authorizations given by the controller to fire a transition; and those prefixed by "`ctr_`" represent transitions that can be triggered by the controller, without any request from the environment. We transform the global model in a synchronous model amenable to validation tools [10]. This transformation is enhanced by taking into account the state-based control in GASPARD.

3. Reactive systems and DCS

3.1. The synchronous approach

Embedded systems, which continuously interact with their environment, and which have strict safety-criticality constraints, are considered as reactive systems. The synchronous approach to reactive systems [6] provides programmers with concrete tools for design: programming languages, compilers, code generators, model-checkers and discrete controller synthesis (DCS) tools [11]. The model-based approach to reactive systems relies on the basic formalism of labeled transition systems. The finite state machines are composed of discrete states, that abstract significant value domains, and characterize significant dynamical sequences of events. Transitions between states are labeled with conditions and actions. Such modeling formalisms have been applied to describe the control behavior of multi-task systems, where multiple modes correspond to different resource management as well as QoS policies. The switches are labeled with conditions that control the adaptation.

The synchronous languages [6] provide high-level language support for the structured construction of such models, enabling the consideration of large and complex systems, the resulting composition being computed by the compilers. Such transition systems can also be viewed under an equational form, as a sequential transition function, as classical Boolean circuits, illustrated in the inner box of Fig. 8. Logical properties of these automata concern typically reachability of states, or invariance of subsets of the state space, required or forbidden sequences of transitions.

The resulting transition system formal model is the concrete representation of designs upon which operations are defined, in the form of algorithms for their analysis (e.g., for verification or test-case generation) or transformation (e.g., automated partitioning into distributed communicating processes). They constitute executable formal models in the sense that code can be generated for efficient simulation purposes, or for execution, in a platform-dependent way. The underlying models, and their related algorithmic tools, are subject to complexity and combinatorial explosion problems. However, the size of manageable models is such that meaningful systems can take advantage of the offered services of analysis. This is especially true when the models are adequately structured, enabling the reactive kernel to be abstracted from the more computational parts of the system [12]. Expressivity of the models can be augmented with quantitative information (concerning time , but also values of variables that can be handled by static analysis, or even hybrid systems). However, this involves a considerable cost w.r.t. the related algorithms, and the size of manageable systems is therefore limited.

3.2. Discrete controller synthesis (DCS)

DCS is one of the automated techniques that can exploit transition system models. It consists in considering on the one hand, the set of possible behaviors of a discrete event system, where inputs are partitioned into uncontrollables (I_u) and controllables ones (I_c), as illustrated in Fig. 8.The uncontrollable inputs typically come from the system's environment, while the values of the controllable inputs are given by the synthesized controller. On the other hand, it requires a specification of a control objective: a property typically concerning reachability or invariance of a state space subset. For instance, behaviors should remain within

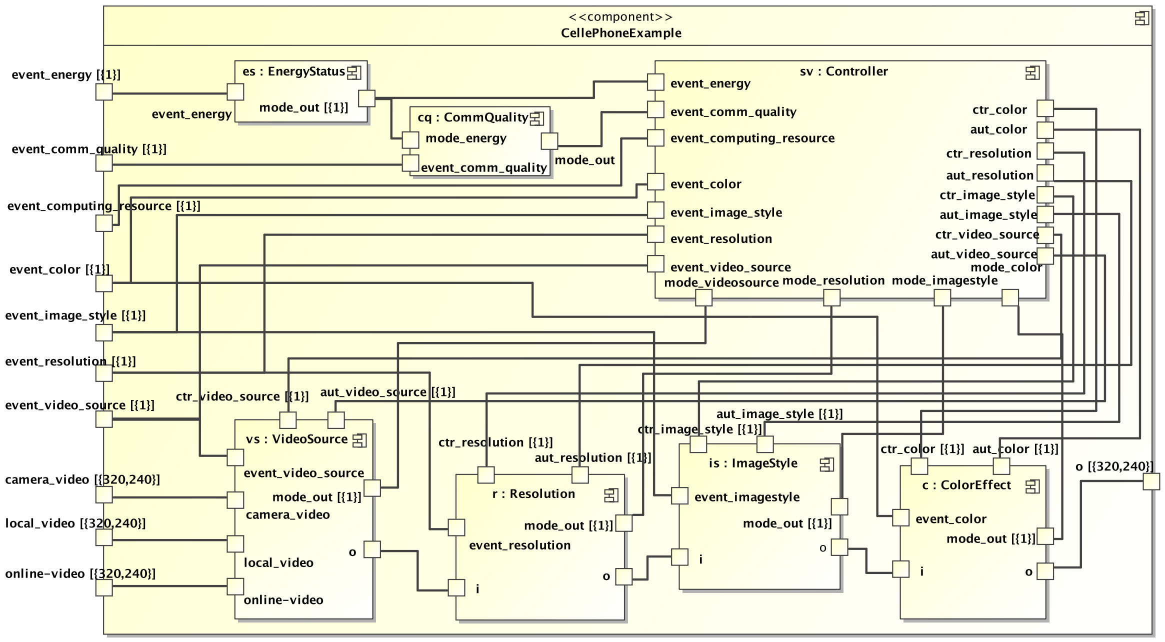


Figure 7. A global model view of the multimedia functionality module.

the subset of states declared safe, or the termination state of the application should always be reachable (i.e., avoid entering subspaces of states from where there is no way to termination). DCS consists of the computation of the necessary constraints on controllable events, w.r.t. the current state and all possible uncontrollable inputs, so that the objective properties are satisfied by the resulting controlled system. These computed constraints yield a *controller* which defines, together with the initial system, a *controlled system*, satisfying the synthesis objectives. When DCS is maximally permissive, the constraint on controllables is minimal: the most possible behaviors are kept. This can be formulated: *whatever the uncontrollable inputs sequences, the controlled behavior satisfies the objectives.*

In Fig. 8, the controller synthesized is the function h. It is automatically computed from the reactive system (X, g, f), X being the state of the system, g the transition function, and f the output function. This controller h is such that, in a given state X, and given any uncontrollable input I_u, give values to controllables I_c, so that the resulting behavior satisfies the control objectives.

DCS was originally defined in the framework of language theory, often called supervisory control of discrete event systems, and is related to game theory. It has been formulated in terms of labeled transition systems, and involves algorithms that explore symbolically the state space in a way like model-checking verification, with complexity issues and capacities of the same order. Within the synchronous approach, DCS has been defined and implemented as a tool integrated with the synchronous languages: SIGALI [11]. It

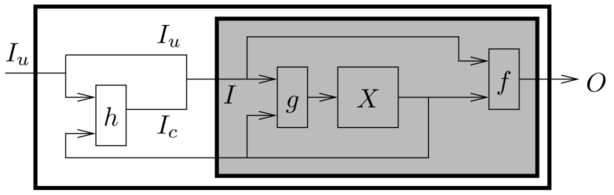


Figure 8. An equational view of a reactive system controlled by h obtained by DCS.

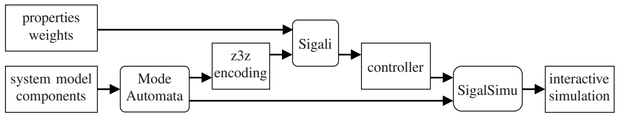


Figure 10. A synchronous tool suite for DCS.

handles transition systems with the multi-event labels typical of the synchronous approach, and features weight functions mechanisms to introduce some quantitative information and perform optimal DCS. It has been applied to thegenration of correct task handlers, and integrated in a domain-specific language [13], making the technique more user-friendly.

In our case study, we use the toolset illustrated in Fig. 10: models of behaviors are synchronous programs in the mode automata language; its compiler produces a format (called `z3z`) taken by SIGALI, which also takes the objectives, and produces a controller. This latter is combined with an expanded form of the mode automaton, for co-simulation.

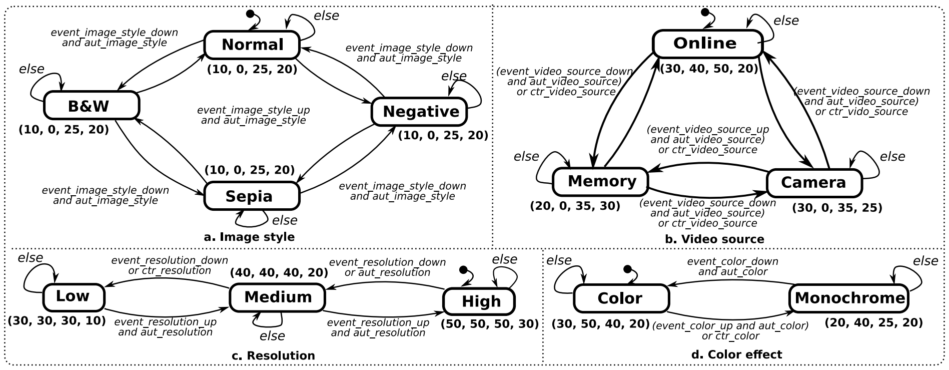

event_image_style_down and aut_image_style
Normal
else
(10, 0, 25, 20)
event_image_style_down and aut_image_style
else
B&W
(10, 0, 25, 20)
event_image_style_up and aut_image_style
else
Negative
(10, 0, 25, 20)
(10, 0, 25, 20)
event_image_style_down and aut_image_style
Sepia
event_image_style_down and aut_image_style
else
a. Image style
Online
else
(event_video_source_down and aut_video_source) or ctr_video_source
(30, 40, 50, 20)
(event_video_source_down and aut_video_source) or ctr_vido_source
(event_video_source_up and aut_video_source) or ctr_video_source
else
Memory
Camera
else
(20, 0, 35, 30)
(event_video_source_down and aut_video_source) or ctr_video_source
(30, 0, 35, 25)
b. Video source
else
event_resolution_down or ctr_resolution
(40, 40, 40, 20)
event_resolution_down or aut_resolution
else
Low
Medium
High
(30, 30, 30, 10)
event_resolution_up and aut_resolution
else
event_resolution_up and aut_resolution
(50, 50, 50, 30)
c. Resolution
else
event_color_down and aut_color
else
Color
Monochrome
(30, 50, 40, 20)
(event_color_up and aut_color) or ctr_color
(20, 40, 25, 20)
d. Color effect

Figure 9. State machines associated with *VideoSource*, *ColorEffect*, *ImageStyle* and *Resolution* component.

4. Application of DCS to our case study

4.1. From GASPARD to synchronous languages

4.1.1. Transformation in synchronous automata.. UML state machines and collaborations are easily translated into synchronous automata or synchronous dataflow equations. UML state machines are very similar to synchronous automata in structure, hence the transformation is direct. UML trigger events on transitions are transformed into mode transition conditions in terms of expressions. A state machine without hierarchy is transformed into one synchronous automaton. A hierarchical state machine is transformed into a parallel and hierarchical composition specification of corresponding automata. Parallel composition is the synchronous one. According to collaborations, different modes modeled as sets of equations, are set directly in the states of mode automata. Apart from the transformation into explicit automata, UML state machines and collaborations can be also transformed in pure equational synchronous programs, where state machines and collaborations are transformed respectively by using `if/then/else` statements and mode invocations according to mode values.

4.1.2. Consumption and Quality of Service.. In our example, the modes defined in the components are characterized by quantitative attributes representing the following non-functional properties: energy consumption (E), communication quality required (CQ), computing resource consumption (CR) and memory consumption (M). This is illustrated in Fig. 9 by the tuple annotations (E, CQ, CR, M) associated with each state denoting a mode. These non-functional requirements are to be understood as instantaneous consumptions of quantitative resources that may vary from one system reaction to another. The values associated locally with the modes are combined additionally when components are composed in parallel, so as to obtain global costs for the whole system from the local costs of its components.

4.2. Adaptation policy and DCS application

Possible behaviors involving the above characteristics are, e.g., that the consumption of a resource must respect the bounds defined by its capacity. Therefore, if a new functionality is executed, then the other tasks that are already running should switch to lower consumption modes, possibly reducing their quality as well. Or, if the level of the battery goes down, then the control should switch task modes so that the lower energy capacity is respected.

Such control strategies are defined by properties expressed in terms of the states and inputs of the system. The SIGALI tool allows one to express Boolean properties on states and inputs ($P : \mathbb{B}^n \rightarrow \mathbb{B}$), and to build cost functions, associating numerical values (here, assumed to be integer, without loss of generality) with Boolean functions of states and inputs ($f : \mathbb{B}^n \rightarrow \mathbb{N}$). We essentially consider *invariance*, by specifying a subset of system states, defined by a Boolean property P: P is invariant for the system if for all states in this subset, transitions from these states lead to states in the same subset. This invariance property of the system is noted $\forall\Box P$: the Boolean property P is true at every instant of every trace of the system.

An example of state property is the exclusivity of two modes from two components. For example, in order to avoid waste of resources, it can be useful to specify that the modes B&W (*ImageStyle* component) and Color (*ColorEffect* component) are never active at the same instant. This invariance property is denoted: $\forall\Box\left(\overline{\text{B\&W} \wedge \text{Color}}\right)$.

Considering invariance properties, the basic DCS operation which we use is to generate the controller in order to *make invariant* the set of states given as objective, w.r.t. the initial state. The controller will constrain the values of the controllable Booleans in such a way that they will inhibit transitions going out of the set, *as well as* transitions going to states from which uncontrollable transitions could lead out of the set. Invariance is a safety property, and is also preserved by synchronous composition, hence allowing incremental controller synthesis: the constraints add up into a controller for all objectives.

Another example involves cost functions: for a function where the global cost is defined by the sum of the local costs of the components, e.g. for memory footprint, there is a bound defined by the size of the memory. Thus, if we note by $f_M = f_{IS} + f_{VS} + f_R + f_{CE}$ the cost function associating with each global state of the system, the memory usage in this state, we can enforce the fact that this usage will always be bounded by the memory available (here, 90 units), by the invariance synthesis objective: $\forall\Box\big(f_M \leq 90\big)$.

The bound itself can vary in time: it is actually the case for the available energy. In this case, we add to our model an automaton which represents the environment, namely here the energy resource available. This automaton can be seen in Fig. 11. We associate a cost function f_{EA} with this automaton, associating with its states the energy quantity instantaneously available. Then, we can bound the energy consumption f_E by the available energy: $\forall\Box\big(f_E \leq f_{EA}\big)$.

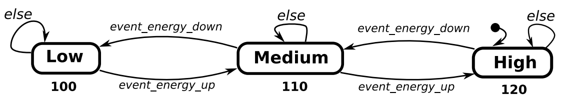


Figure 11. Energy resource model

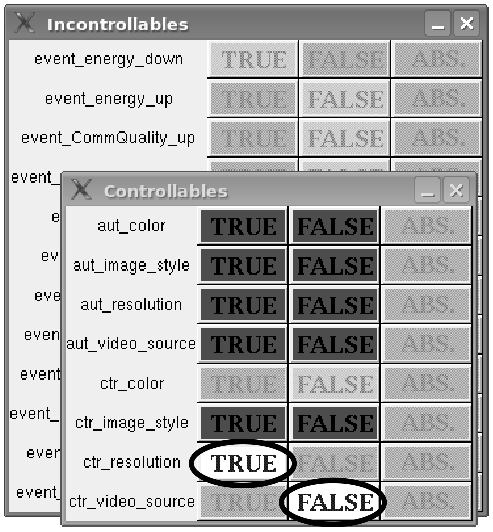
It is also interesting to define conditioned objectives, e.g., according to the battery status, which will lead to adaptive strategies. An example of such adaptive strategy is to forbid costly modes in medium or low battery states, so as to limit in time the energy consumption. We identify here two modes, high resolution and color mode, which will be enforced exclusive when not in high battery state:
$\forall\Box\Big(\big(\mathrm{High}_{\mathrm{Res}} \wedge \mathrm{Color}\big) \Rightarrow \mathrm{High}_{\mathrm{Energy}}\Big)$.

Finally, a one-step optimal synthesis operation is available in order to control the system so as to choose the next configuration where cost functions are minimized (e.g., power) or maximized (QoS). The last synthesis objective constrains the controller, among several transitions satisfying the preceding invariance properties, to choose the transition leading to the state that consumes the minimum energy quantity: $\min_{q \rightarrow q'}(f_E(q'))$.

With these DCS operations, different policies or strategies can be obtained *automatically* by changing the objectives, hence providing for separation of concerns and making the models easy to reuse.

4.3. Simulation

The controller computed by SIGALI is extracted, and co-simulated with the system with the SIGALSIMU tool. Fig. 12 shows a particular simulation step, where the controller enforces the values of two controllable inputs so as to keep the properties satisfied. At this step, the system is in high energy, high resolution, and color state. We then simulate



ctr_resolution [✓ TRUE] FALSE ABS.
ctr_video_source TRUE [✓ FALSE] ABS.

Figure 12. Simulation of the controlled system

the discharge of the battery by the occurrence of the uncontrollable input `event_energy_down`. On the controllable inputs panel, the clearer inputs shown with ellipsis are those whose values have been forced by the controller. It is here the case of the input `ctr_resolution`, meaning that the controller has triggered the transition from high to medium resolution state.

5. Discussions

There are several studies dealing with the safe design of systems modeled in UML. In [14], [15], the authors concentrate on the verification of systems in which the control is described with UML state machines and collaborations, by using model-checking. In [16], the focus is put on the definition of a UML profile and a toolset for the modeling, verification and simulation of real-time and embedded systems. The verification relies on model-checkers and a theorem prover. Our approach shares several common features with all these studies: UML-based modeling of systems, connection with formal validation tools, etc. In our case, the SIGALI tool allows for model-checking and the compilers of synchronous languages are usable for verification.

However, our proposition differs from the above studies in that it specifically considers high-performance embedded systems in a more general design framework, GASPARD, adopting the MARTE standard profile for modeling, and MDE techniques for model refinements towards different design exploration technologies. In particular, it targets the synchronous technology, which gives access to a wide range of formal validation tools and techniques. Here, we chose DCS instead of verification for its constructiveness.

DCS techniques have been studied quite less than verification and model-checking, because they seem to involve a design approach closer to control theory than to computer programming, with a model of the system and its behaviors separately from its control. Applications of DCS have classically concerned discrete control theory problems in manufacturing systems. However, variants of controller synthesis have been applied to timed automata for application-specific scheduling of task systems, and job-shop scheduling problems [17], [18]. Contrarily to model-checking, DCS is more constructive, in that instead of diagnosing bugs, it produces (when possible) a correct solution. It finds this solution even though it is so intricate that a programmer would not have thought of it, unless for hours or days of tedious and boring work, involving cycles of design, validation, and back while a controller performed by hand is not correct. In addition, its mechanical nature makes it integrable in tools where final users do not need any expertise in formal techniques.

6. Concluding remarks

In this paper, we showed how discrete controller synthesis is used in the model-driven engineering environment GASPARD, to allow for the safe design of data-intensive embedded systems. The proposed approach is illustrated on a multimedia application module, which is first modeled using the OMG MARTE standard profile. After the translation of the resulting models in synchronous languages, we focused on the control part in order to *automatically* synthesize a global application controller that satisfies safety properties w.r.t functional and non functional requirements.

Perspectives to our work include: the applicability to larger case studies in order to address more the scalability of the approach; the use of techniques of modular discrete controller synthesis to favor the scalability; and also the reusability of correctly controlled components in new system designs that will take advantage of their correctness.

References

[1] A. Gamatié, S. Le Beux, E. Piel, A. Etien, R. Ben Atitallah, P. Marquet, and J.-L. Dekeyser, "A model driven design framework for high performance embedded systems," INRIA, France, Research Report 6614, August 2008, http://hal.inria.fr/inria-00311115/en.

[2] The DaRT Project, "GASPARD: Graphical Array Specification for Parallel and Distributed computing," 2008, http://www2.lifl.fr/west/gaspard/index.html#g2.

[3] OMG Group, "Modeling and analysis of real-time and embedded systems (MARTE)," 2007, www.omgmarte.org/.

[4] P. Boulet, "Formal semantics of Array-OL, a domain specific language for intensive multidimensional signal processing," INRIA, France, Research Report 6467, March 2008, http://hal.inria.fr/inria-00261178/en.

[5] A. Demeure and Y. Del Gallo, "An array approach for signal processing design," in *Sophia-Antipolis Conf. on Micro-Electronics (SAME'98), France*, Oct. 1998.

[6] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone, "The synchronous languages twelve years later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, Jan. 2003, special issue on embedded systems.

[7] F. Maraninchi and Y. Rémond, "Mode-automata: a new domain-specific construct for the development of safe critical systems," *Sci. Comput. Program.*, vol. 46, no. 3, 2003.

[8] J.-L. Colaço, G. Hamon, and M. Pouzet, "Mixing signals and modes in synchronous data-flow systems," in *ACM International Conference on Embedded Software*, Oct. 2006.

[9] A. Gamatié, E. Rutten, and H. Yu, "A model for the mixed-design of data-intensive and control-oriented embedded systems," INRIA, Research Report 6589, 2008, http://hal.inria.fr/inria-00293909/fr.

[10] H. Yu, A. Gamatié, E. Rutten, and J.-L. Dekeyser, "Model transformations from a data parallel formalism towards synchronous languages," in *Embedded Systems Specification and Design Languages, Selected Contributions from FDL'07 Series*, ser. Lecture Notes in Electrical Engineering, V. Eugenio, Ed., vol. 10. Springer Verlag, 2008.

[11] H. Marchand, P. Bournai, M. L. Borgne, and P. L. Guernic, "Synthesis of discrete-event controllers based on the Signal environment," *Discrete Event Dynamic System: Theory and Applications*, vol. 10, no. 4, pp. 325–346, Oct. 2000.

[12] K. Altisen, A. Clodic, F. Maraninchi, and E. Rutten, "Using controller-synthesis techniques to build property-enforcing layers," in *Proc. of the European Symp. on Programming, ESOP'03,* April 7 - 11, 2003,Warsaw, Poland, 2003, pp. 174–188, lNCS nr. 2618.

[13] G. Delaval and E. Rutten, "A domain-specific language for multi-task systems, applying discrete controller synthesis," *Journal on Embedded Systems, special issue on Synchronous Paradigm in Embedded Systems*, vol. Volume 2007, 2007.

[14] D. Latella, I. Majzik, and M. Massink, "Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-Checker," *Formal Aspects Computing*, vol. 11, pp. 637–664, 1999.

[15] T. Schäfer, A. Knapp, and S. Merz, "Model checking UML state machines and collaborations," in *CAV Workshop on Software Model Checking*, ser. ENTCS 55(3), Paris, France, 2001.

[16] S. Graf, "Omega – correct development of real time embedded systems," *SoSyM, int. Journal on Software & Systems Modelling*, vol. 7, no. 2, 2008.

[17] C. Kloukinas and S. Yovine, "Synthesis of safe, QoS extendible, application specific schedulers for heterogeneous real-time systems," in *15th Euromicro Conference on Real-Time Systems (ECRTS'03)*, Porto, Portugal, Jul. 2003.

[18] K. Altisen, G. Gößler, and J. Sifakis, "Scheduler modeling based on the controller synthesis paradigm," *Real-Time Syst.*, vol. 23, no. 1-2, pp. 55–84, 2002.