HAL
open science

A Design Method for Synthesizing Control-Command Systems out of Reusable Components

Salam Hajjar, Emil Dumitrescu, Laurent Pietrac, Eric Niel

▶ **To cite this version:**

Salam Hajjar, Emil Dumitrescu, Laurent Pietrac, Eric Niel. A Design Method for Synthesizing Control-Command Systems out of Reusable Components. IFAC IWDES, May 2014, Cachan, France. 10.3182/20140514-3-FR-4046.00111 . hal-01080076

HAL Id: hal-01080076

https://hal.science/hal-01080076v1

Submitted on 4 Nov 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Design Method for Synthesizing Control-Command Systems out of Reusable Components

Salam HAJJAR, Emil DUMITRESCU, Laurent PIETRAC, Eric NIEL

Université de Lyon, INSA Lyon, Ampère (UMR5005), F-69621 Villeurbanne, France (e-mail: firstname.lastname@insa-lyon.fr).

Abstract: This paper investigates an industrial design issue related to code reusability: building control-command systems out of Commercial off the shelf (COTS) components. The design method proposed uses in synergy the formal verification (FV) and the discrete controller synthesis (DCS) techniques. COTS are formally specified using temporal logic and/or executable observers, and coded according to their formal specification. New functions are built by assembling COTS together. The COTS assembly operation is not error free: the resulting assembly may not achieve the desired function it is supposed to. For these reasons, COTS assemblies need to be formally verified and if errors are found, they must be corrected using DCS. The resulting system is ready for hardware (e.g. FPGA) implementation.

Keywords: Formal verification, discrete controller synthesis, COTS, simulation, embedded systems, control-command.

1. INTRODUCTION

Due to design constraints bounding delays, costs and engineering resources, component re-usability has become a key issue in embedded design. The expertise of the design process has been shifted from code writing to the efficient management of Commercial off the Shelf (COTS) libraries. By assembling adequately COTS components, new functions can be quickly built. Paradoxically, the brute force application of this method has lead to important design and maintenance costs, far from the theoretically expected gains. This is caused by an antagonism between the genericity expected for a COTS, and the context-specific needs which fail to be handled correctly by that COTS. Indeed, by assembling COTS which have been separately designed, the resulting interactions cannot be entirely anticipated. Thus, unwanted behaviors may occur, although each component taken separately is considered free of errors. Ensuring a safe behavior of the COTS-based system is an important challenge. It calls for safe design methods and techniques, ensuring functional correctness. Besides simulation, the model checking technique (Clarke, 2008) is vital for discovering subtle bugs, which are very difficult to uncover by simulation. Even though this technique has become mature, the designer must correct errors manually, which is an error-prone task: by attempting to manually correct an error, another error is introduced, which creates a vicious circle. This situation emphasizes the need to complete the automatic error detection, by an automatic error correction.

This work advocates the use of the Discrete Controller Synthesis (DCS) technique (Marchand et al., 2000) in order to generate correct-by-construction design code. The design method proposed in this paper highlights the synergy and the interdependence between these formal tools for achieving control-command COTS-based design.

The development of component-based design has been dependent on the growing maturity of formal techniques. (Addy and Sitaraman, 1999) have proposed the formalization of the COTS interface, in order to facilitate their composition. A similar formalization is proposed by (De Alfaro and Henzinger, 2001) with the interface automata, capturing compositional aspects such as environment assumptions. Interface generation is also considered by (Roop et al., 2009), in order to solve mismatches between interacting protocols. A formalization of the COTS behavior has been proposed by (Guerrouat and Richter, 2005), using extended finite state automata. On a more practical point of view (Abts, 2002) show that COTS-based design faces in general exponential blowup of maintenance costs. This phenomenon is due to the lack of control on the COTS behaviors, which are handled as black box components. (Xie et al., 2007) show the importance of the model checking technique, together with the assume-guarantee reasoning in COTS-based design. The DCS has been suggested by (Altisen et al., 2003) to synthesize properties-enforcing layers, on the composition of local robot controllers. A contract-based modular design approach has also been proposed by (Delaval et al., 2010). This approach also relies on DCS and provides a toolchain able to build controlled systems automatically.

This paper takes over the issues presented by (Hajjar et al., 2013), and tackles two context-specific concerns related to the application of DCS. On the one hand, hardware target implementations require a particular representation

of the synthesized controller. On the other hand, the existence of interfaces between control-command components, requires additional care: sometimes, generated controllers become part of the interface between two or more COTS and their behavior should not contradict the behavior expected for that interface. As it is shown in the sequel, this requirement cannot be handled by DCS alone. The control solution needs to be formally verified. These are the specificities of this work, compared to other existing propositions, namely (Delaval et al., 2010). The validity of this approach is demonstrated on an industrial case study concerning a train control-command system.

The rest of the paper is organized as follows: section 2 recalls the backgrounds of the models and techniques used throughout the method proposed in this paper. Section 3 highlights the structural issues in applying DCS to hardware designs. Section 4 presents a variant of the DCS technique, taking into account environment assumptions. The COTS-based design method is presented in section 5. Section 6 presents the controller validation issues. Section 7 illustrates this design method on an industrial design.

2. BACKGROUND AND DEFINITIONS

The Boolean Finite State Machine (BFSM). This model is very useful in our context because it is structurally and dynamically close to the hardware control-command systems we handle. Indeed, these are composed of Boolean variables, implementing either inputs, states or outputs. Thus the Boolean FSM is defined as a tuple $M = \langle q_0, X, Q, \delta, PROP, \lambda \rangle$, where q_0 designates the initial state, X a set of Boolean inputs, Q is the set of states of M, $\delta : Q \times X \to Q$ is the transition function, $PROP$ is a set of atomic Boolean propositions, and $\lambda : Q \to \mathbb{B}^{|PROP|}$ is a labelling function modeling the outputs of M. This formal model is automatically extracted from design code written in VHDL, or in a similar proprietary framework. These programs feature systematically a hardware clock, which triggers all the transitions of the design, and which is considered to be common to the whole design. Under these circumstances, the clock representation can be left implicit inside the formal model.

Formal Requirement Specifications. Formal specifications are expressed either logically, as temporal logic formulæ, written in the PSL (IEEE, 2005) standard language, or operationally, as a "program" modeled formally by a BFSM and referred to as a *monitor*.

Control-command COTS are the basic building blocks considered in this work. A stand-alone COTS component C is defined as a 4-tuple $C = \langle I^C, M^C, A^C, G^C \rangle$, where I^C is the COTS' input-output interface, M^C is the behavioral model of the COTS expressed as a BFSM, A^C is a set of assumptions on the expected behavior of the environment of C and G^C is a set of guarantees on the behavior M^C of C. Both the assumptions and the guarantees are expressed formally, either as PSL formulæ or as monitors. A COTS C satisfies a guarantee $g \in G^C$ provided an assumption $a \in A^C$ holds. This is denoted:

$$M^C, \langle a \rangle \models g$$

A COTS is considered rather a "mature" component than a "perfect" one; it probably has hidden bugs, and building designs out of existing COTS elements also amounts to mixing unwanted behaviors from each building block. The COTS's behavior is expressed as design code, using a standard and/or proprietary framework. All the components handled in this work are automatically translatable into BFSMs.

COTS assembly. This is the act of composing COTS components together, in order to produce a new behavior. This operation produces a new component which is not considered as a COTS until its maturity is assessed. The assembly operation produces new sets of assumptions and guarantees: assumptions can be implied by newly added guarantees and need not be assumed anymore. They can also be contradicted by newly added guarantees, in which case they cannot be assumed anymore. These issues are not developed in this paper. The behavior of a COTS assembly is given by the synchronous composition operation, denoted $||$, between their corresponding behavioral models.

The discrete controller synthesis (DCS) This technique enforces the satisfaction of a safety requirement P on a given BFSM model M by attempting to make invariant the greatest subset of states of M which satisfy P. The input set of M is divided into two disjoint subsets: controllable X_c and uncontrollable X_{uc} inputs. The target set satisfying P is made invariant by disabling all the transitions of M leading out of it. This is achieved by generating a supervisor, which assigns adequate values to the controllable inputs X_c.

The DCS proceeds in two steps (Marchand et al., 2000): (1) computation of the invariant under control (IUC) set and (2) computation of the supervisor. The computation of IUC calls recursively a basic step: finding the set of controllable predecessors of a given set of states $E \subseteq Q$. This step is implemented by the $CPRED$ operator:

$$CPRED(E, \delta) = \{q \in Q \mid \forall \mathbf{x_{uc}} \in \mathbb{B}^{|X_{uc}|}, \exists \mathbf{x_c} \in \mathbb{B}^{|X_c|}, \exists q' \in E : q' = \delta(q, \mathbf{x_{uc}}, \mathbf{x_c})\}$$

In other words, the state q is a controllable predecessor of a state $q' \in E$ iff for any uncontrollable value x_{uc}, there exists a controllable value x_c such that the transition function δ leads to q'.

The resulting invariant under control set $\mathcal{IUC}$ is the fixed point of the equation:

$$IUC_0 = \{q \mid P \text{ is true in } q\}$$
$$IUC_{i+1} = IUC_i \cap CPRED(IUC_i, \delta)$$

A supervisor does not exist if the $\mathcal{IUC}$ set is empty or if it does not contain q_0. When it exists, the supervisor is defined as: $SUP = \{(q, \mathbf{x_{uc}}, \mathbf{x_c}) \mid \delta(q, \mathbf{x_{uc}}, \mathbf{x_c}) \in IUC\}$.

3. DCS FOR HARDWARE DESIGN

The supervisor provided by DCS is implemented as a characteristic function:

$$\mathcal{SUP} : Q \times \mathbb{B}^{|X_{uc}|} \times \mathbb{B}^{|X_c|} \to \mathbb{B}$$

defined as:

$$\mathcal{SUP}(\mathbf{q}, \mathbf{x_{uc}}, \mathbf{x_c}) = 1 \text{ iff } (\mathbf{q}, \mathbf{x_{uc}}, \mathbf{x_c}) \in SUP$$

The actual control of M requires solving the equation

$$\mathcal{SUP}(\mathbf{q}, \mathbf{x_{uc}}, \mathbf{x_c}) = 1$$

continuously, for each reaction of M, considering $\mathbf{X_c}$ as unknown variables. However, a hardware (FPGA) implementation of this control loop requires that the value of each controllable variable x_c be computed by an appropriate expression, for each reaction of M. The supervisor decomposition technique presented in (Dumitrescu et al., 2008) is used in order to obtain systematically the control architecture presented in Figure 1. Let $exp|_{x \leftarrow 0/1}$ denote the negative/positive co-factor of a Boolean expression exp with respect to a variable x. The supervisor $\mathcal{SUP}$ is recursively decomposed according to the rule:

$$f_i = \neg \mathcal{SUP}_i|_{x_{ci} \leftarrow 0} \wedge \mathcal{SUP}_i|_{x_{ci} \leftarrow 1} \vee$$
$$x_{ci}^{env} \wedge \mathcal{SUP}_i|_{x_{ci} \leftarrow 0} \wedge \mathcal{SUP}_i|_{x_{ci} \leftarrow 1}$$

where $\mathcal{SUP}_0 = \mathcal{SUP}$ and $\mathcal{SUP}_i = \mathcal{SUP}_{i-1}|_{x_{ci-1} \leftarrow f_{i-1}}$.

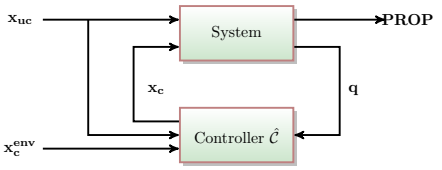


Fig. 1. Target control architecture for hardware designs

The resulting controller is a vector $\hat{\mathcal{C}}$ of m Boolean functions, where m is the number of controllable variables:

$$\hat{\mathcal{C}} = \begin{pmatrix} f_1(q, x_{uc}, x_{c1}^{env}, f_2, \ldots, f_m) \\ f_2(q, x_{uc}, x_{c2}^{env}, f_3, \ldots, f_m) \\ \ldots\ldots \\ f_m(q, x_{uc}, x_{cm}^{env}) \end{pmatrix}$$

This decomposition process associates each controllable variable x_{ci} with an auxiliary variable x_{ci}^{env}. The action of $\hat{\mathcal{C}}$ is similar to filtering: at each moment, depending on the current state q and on $\mathbf{x_{uc}}$, x_{ci} is assigned either x_{ci}^{env} or $\neg x_{ci}^{env}$. The auxiliary variables x_c^{env} are meant to be driven from outside $M||\hat{\mathcal{C}}$ in order to specify desired values for x_c. This is why they are referred to as *environment variables*.

It is worth noting that this control paradigm is totally opposite to the one developed by the supervisory control theory (Ramadge and Wonham, 1989). Indeed, unlike conventional supervisors, the control architecture presented above *interferes with the environment* by filtering if needed the values of the controllable inputs. Obviously, this situation is globally undesirable but acceptable for control reasons; however, this issue induces additional design constraints, developed in Section 5.

A DCS illustrative example Consider the state-based design illustrated in Fig 2. Let a property $P = always\neg(E_1 \vee E_2)$ be the target requirement to enforce using DCS, by controlling the input variable go. The IUC computation algorithm gives the following results: $IUC_0 = \{A, B, C\}$, $IUC_1 = \{A, C\}$, $IUC_2 = \{A, C\}$. The final IUC set is $\{A, C\}$. The generated controller $\hat{\mathcal{C}}$ assigns the controllable variable go, so that the controlled system always remains inside the set of states IUC, as illustrated in Figure 3.

4. THE ENVIRONMENT-AWARE DCS (EDCS)

In the context of COTS-based design, environment assumptions are of great importance. It often happens that

Fig. 2. A 5-states design to be controlled using DCS

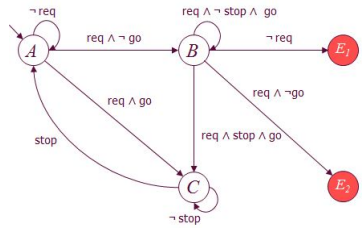
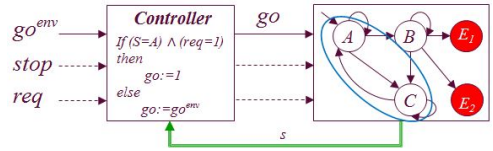


Fig. 3. The controlled 5-states system



the environment of a given COTS consists of a collection of other COTS. Thus, a direct connection between a COTS and the "physical world", made of sensors and of actuators, may not always exist. In this situation, unlike a physical environment, the environment of a COTS must feature a precise behavior so that the COTS at hand can fulfill its function.

The conventional DCS algorithm used in this work does not support the specification of environment assumptions. In order to handle this additional information, a variant of the DCS algorithm is proposed, called environment aware DCS (EDCS). It redefines the computation of the controllable predecessors, by assuming that at each step, the uncontrollable inputs satisfy the environment assumptions. Each assumption is modeled as a safety property $a^C \in A^C$ concerning the uncontrollable inputs. It is translated into an invariant $\mathcal{A} : Q \times \mathbb{B}^{|X_{uc}|} \rightarrow \mathbb{B}$, defined as the set of all the transitions of M satisfying a^c:

$$\mathcal{A}(q, \mathbf{x_{uc}}) = \{\exists \mathbf{x_c} : a^c \text{ is true in state } \delta(q, \mathbf{x_c}, \mathbf{x_{uc}})\}$$

which is supposed to be always true. The computation of the environment-ware controllable predecessors is defined as follows:

$$CPRED^{env}(E, \delta, \mathcal{A}) = \{q \in Q \mid \forall \mathbf{x_{uc}} \in B^{|X_{uc}|},$$
$$\exists \mathbf{x_c} \in B^{|X_c|}, \exists q' \in Q :$$
$$(q' = \delta(q, \mathbf{x_{uc}}, \mathbf{x_c}) \wedge \mathcal{A}(q, \mathbf{x_{uc}})) \rightarrow q' \in E\}$$

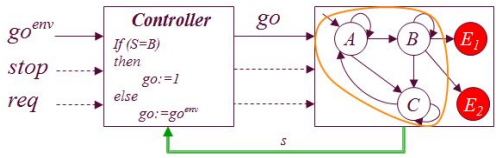
The recursive application of $CPRED^{env}$ produces an invariant under control set under an environment assumption. This rule is less "pessimistic" with respect to the uncontrollable input variables, and thus less restrictive. The resulting IUC set is often larger than the one obtained with conventional DCS.

EDCS illustrative example For a sample environment assumption stating that the uncontrollable req must be asserted when the system is in state B: $always(B \rightarrow req)$, the EDCS application computes the invariant under control $IUC = \{A, B, C\}$. Unlike conventional DCS, state B is not pruned, as req is supposed to be asserted whenever this state is active, and thus, due to this assumption, the error state E_1 is not reached.

5. THE SAFE COTS-BASED DESIGN METHOD

The method proposed relies on the conjunction between traditional design techniques, such as simulation and for-

Fig. 4. Controller synthesis using EDCS

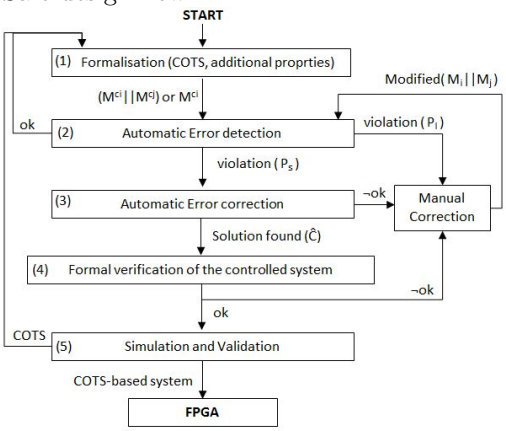


mal verification and the DCS. An overview of the design flow is presented in figure 5. It highlights the conventional steps, like COTS formalization and verification using assume-guarantee reasoning. This paper only focuses on the use of DCS in order to correct a COTS assembly, shown at step 3, and the verification of the controlled system, shown at step 4.

The DCS application occurs after the failure of the formal verification (step 2) applied to a safety requirement. The most delicate operation here, is the construction of the controllable input set, intended to be assigned by the controller. The controllable candidates are supplied by the model-checking counterexample. The designer chooses among these candidates, but should avoid controlling inputs driven by sensors, or inputs carrying data.

The formal verification of the controlled system occurs at step 4. It has two main motivations. As the generated controller interferes with the environment of the controlled system, it can be in contradiction with the environment assumptions of this system. Safety environment assumptions can be taken into account by EDCS, but liveness environment assumptions cannot. Thus, the first objective of this verification step is to ensure that the liveness environment assumptions are not broken, and thus, that the liveness guarantees are preserved. Second, it must be ensured that the controller satisfies application-specific requirements. These are developed in the next section.

Fig. 5. Safe design flow

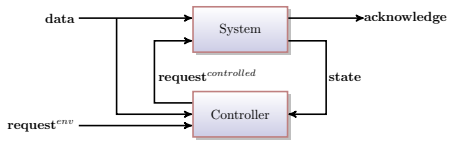


6. IMPLEMENTATION OF THE CONTROL LOOP

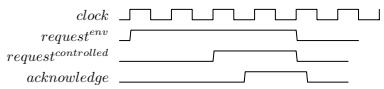
Even though the general control loop architecture presented in Section 3 fills the structural specific needs of hardware design, there remain specific behavioral constraints that need to be guaranteed, and which cannot be handled by (E)DCS.

Controllable inputs with soft reactive constraints. Consider the 4-phase handshake protocol, which is both a generic and representative mechanism, widely used for data exchange and synchronization between hardware components. It is implemented by a pair of Boolean signals: a request and an acknowledge. The handshake protocol starts when the request is activated. The acknowledge is then activated, followed by the request de-activation, and finally by the acknowledge deactivation. This sequence is called a transaction; it achieves a synchronisation between two components. Typically, *transactions have an arbitrary delay.* It is only required that *they last a finite time.* For this specific case, the desired control should implement

Fig. 6. Controlling transactions



(a) Controlled transaction architecture



(b) Desired controller behavior

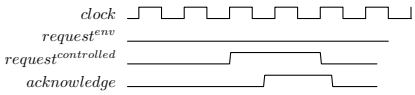
one among the two following behaviors: either prevent a transaction from starting if its beginning is likely to break the requirement to enforce, or let it start otherwise. The transaction start is triggered by the environment, through the $request^{env}$ input variable. This value should be either forwarded or delayed by the controller, as shown in figures 6 a and b: the transaction initiated by the environment is delayed for two clock cycles, and then it is forwarded.

The "event invention" phenomenon. The signal activation and deactivation notions need to be associated to actual Boolean values, usually 1 for the active value and 0 for the inactive value. However, a controller which acts upon the $request^{controlled}$ input is not aware of the notions of activation or deactivation, but only manipulates the values 1 or 0 of this input. At some moments, the value 1 can be forbidden (and thus the value 0 is forced), or vice-versa. However, these two situations are not symmetric: in the first case, the transaction may not start, while in the second, the transaction is forced, or "invented"! This is illustrated in figure 7 a. Usually, transactions also carry data, and hence such a situation does not make sense. Obviously, this is unacceptable. This requires to make sure a posteriori that the controller never "invents" transactions, and if it does, invalidate the control solution.

Hence it is vital to formally ensure the absence of "event-invention" phenomena. The expression of this requirement for a controllable variable x needs to mention systematically x^{env}, which is generated by (E)DCS, as explained in Section 3. However, the variable x^{env} does not exist at the moment DCS starts. It is not possible mention its name to express requirements over the resulting controller. This is why the event invention phenomenon cannot be forbidden, but only detected by model checking.

Detection of "event inventions". This behavior is simply checked by the PSL property:

Fig. 7. The event "invention" phenomenon



(a) Transaction "invention" by the controller

$$NO_EVENT_INVENTION : always \neg(request^{controlled} \wedge \neg request^{env})$$

In complement, it must also be established that the controller is not too restrictive, by delaying transactions forever. Thus, once a transaction starts, it must be acknowledged within finite time:

$$FINITE_TRANSACTION = always(request^{env} \rightarrow eventually\ acknowledge)$$

In order to prove these requirements, it can be needed to assume that once the environment asserts the input request $request^{env}$, it is held until it is acknowledged:

$$REQUEST_STABLE : always(request^{env} \rightarrow next\ (stable(request^{env})\ until\ acknowledge))$$

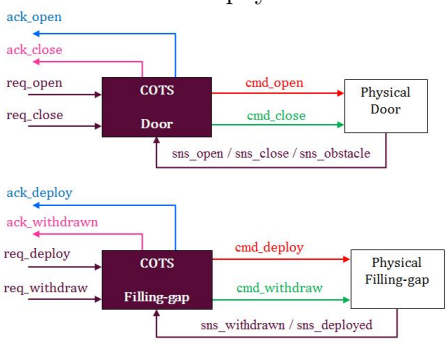
where *stable* is a built-in PSL operator: $stable(x)$ evaluates to true in every cycle where variable x did not change its value with respect to the previous cycle.

If this verification step is successful, the resulting controlled COTS can be considered as valid.

7. INDUSTRIAL APPLICATION

The method proposed above has been applied on a train passenger access control-command system, featuring two COTS: the Door and the Filling gap, as shown in figure 8. This case study has been provided by Bombardier Transport, and has been used during the FerroCOTS project (Jadot, 2009). The Door COTS Model is 4-tuple

Fig. 8. The COTS and their physical environment



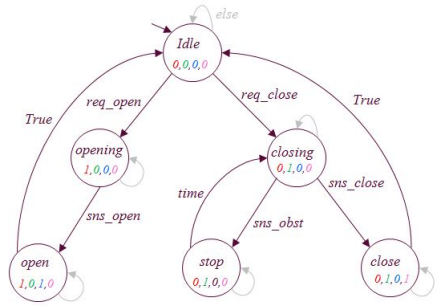
$C^d = \langle I^d, M^d, A^d, G^d \rangle$ where

$$I^d = \{req_open, req_close, sns_open, sns_close, sns_obst, cmd_open, cmd_close, ack_open, ack_close\}.$$

The behavior M^d of the Door COTS is modeled by a BFSM shown in Figure 9. The output values are assigned in each state. The train conductor issues open or close requests via the *req_open* or *req_close* signals and the control command answers with a corresponding acknowledge, once the physical part has had the expected reaction.

The preconditions required for the correct behavior of the Door COTS are given by the set $A^d = \{a_1^d, a_2^d, a_3^d, a_4^d\}$, where $a_{1,2}^d$ express sensor liveness and $a_{3,4}^d$ the absence of request cancellation: $a_1^d = always\ eventually\ (sns_open)$ and $a_2^d = always\ eventually\ (sns_closed)$; $a_3^d = always\ req_open \rightarrow stable(req_open)\ until\ ack_open$ The guarantees of the Door COTS are given by the set $G^d = \{g_1^d, a_2^d\}$. They express the fact that if the door is requested to open or close the request is finally treated. They are modeled by the PSL assertions $g_1^d = always\ req_open \rightarrow eventually\ (ack_open)$; $g_2^d = always\ req_close \rightarrow eventually\ (ack_close)$. The relationship between A^d and G^d are the following:

Fig. 9. Door COTS behavioral model



$$M^d, \langle a_1^d, a_3^d \rangle \models g_1^d \qquad M^d, \langle a_2^d, a_4^d \rangle \models g_2^d$$

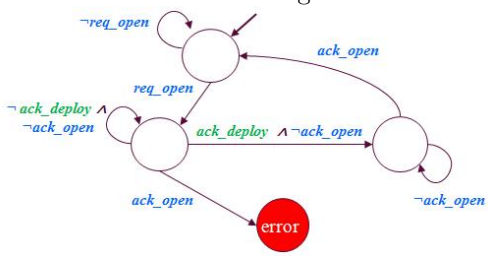
The Filling gap COTS is modeled similarly: $C^{fg} = \langle I^{fg}, M^{fg}, A^{fg}, G^{fg} \rangle$. Its function consists in deploying and retracting the physical filling gap according to the requests sent by the conductor via the inputs *req_deploy* and *req_withdraw*.

The door/filling-gap assembly $C^d || C^{fg}$ is modeled as follows: $I^{asm} = \{I^d \cup I^{fg}\}$, $A^{asm} = \{A^d \cup A^{fg}\}$, $G^{asm} = \{G^d \cup G^{fg}\}$, $M^{asm} = M^d || M^{fg}$. It must implement an additional requirement, expressing the coordination between the door and filling gap operation for security reasons. This requirement states that after an open request, the filling-gap should always deploy before the door is open:

$$P^{asm} : \text{always } req_open \rightarrow ack_deploy \text{ before } ack_open$$

This safety property is modeled as a monitor M^{Pasm} illustrated in figure 10. The COTS assembly does not satisfy P^{asm} and the model checking tool provides a counter-example which highlights the fact that the inputs *req_open*, *req_close*, *req_deploy*, *req_withdraw* are responsible for this violation. By designating these inputs as controllable, P^{asm} is enforced on M^{asm} by EDCS. The resulting control architecture is shown in figure 11. The validation of the controller corresponds to the step 4 of the design method. It consists of formally verifying the

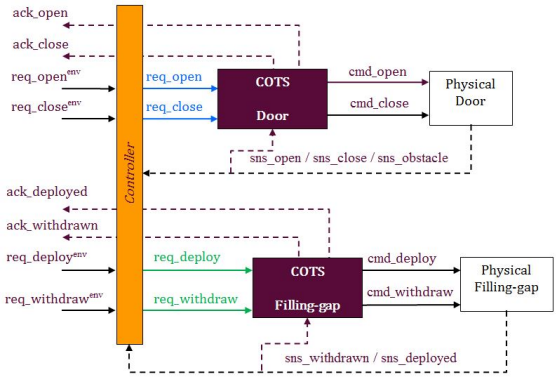
Fig. 10. Monitor M^{Pasm} modeling P^{asm}



properties mentioned in Sections 5 and 6: the guarantees G^d and G^{fg} should still hold after the addition of the controller; on the other hand, "event inventions" should never occur, and the controller should not delay input transactions forever. The absence of event invention is checked by the properties: *never* $(req_\bullet \wedge \neg req_\bullet^{env})$ for the *open* , *close*, *deploy* and *withdraw* mechanisms.

Finite transactions are checked by the properties: *always* $(req_\bullet^{env} \rightarrow eventually)\ ack_\bullet$, which in this particular example happen to be identical to the sets G^d and G^{fg}. Since all these properties are verified, it can be concluded that the controller enforcing P^{asm} is valid. An overview of

Fig. 11. The controlled passengers' access system



the resulting behavior is presented in the simulation trace figure 12. It can be noticed that at simulation time 10sec the designer requests to open the door while the filling-gap is not yet deployed. The controller filters this request until the second 31 where the filling-gap sensor provides the full deploy information and from that moment the controller stops filtering the door opening request. Regardless of the order in which the driver requests the doors and filling gap operations, they always operate in the safe order.

Fig. 12. Simulation of the controlled system

8. CONCLUSION

This paper has presented a safe design method for COTS-based hardware embedded systems. This method uses in synergy the Discrete Controller Synthesis and formal verification techniques in order to produce correct by construction COTS-based systems. Specific issues related to the use of DCS in the hardware design context have been identified and addressed: the structural compatibility between the controller and the system to control, the integration of environment assumptions in solving the DCS problem, and the behavioral validation of the controller. Future directions of this work aim at using DCS for interface generation, as well as handling liveness requirements enforcement.

REFERENCES

Abts, C. (2002). Cots-based systems (cbs) functional density – a heuristic for better cbs design. In *Proceedings of the First International Conference on COTS-Based Software Systems*, ICCBSS '02, 1–9. Springer-Verlag, London, UK, UK.

Addy, E.A. and Sitaraman, M. (1999). Formal specification of cots-based software: a case study. In *Proceedings of the 1999 symposium on Software reusability*, SSR '99, 83–91. ACM, New York, NY, USA.

Altisen, K., Clodic, A., Maraninchi, F., and Rutten, E. (2003). Using controller-synthesis techniques to build property-enforcing layers. In *Proceedings of the 12th European conference on Programming*, ESOP'03, 174–188. Springer-Verlag, Warsaw, Poland. ACM ID: 1765727.

Clarke, E.M. (2008). 25 years of model checking. chapter The Birth of Model Checking, 1–26. Springer-Verlag, Berlin, Heidelberg.

De Alfaro, L. and Henzinger, T.A. (2001). Interface automata. *SIGSOFT Softw. Eng. Notes*, 26(5), 109–120.

Delaval, G., Marchand, H., and Rutten, E. (2010). Contracts for modular discrete controller synthesis. In *Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems*, LCTES '10, 57–66. ACM, Stockholm, Sweden.

Dumitrescu, E., Ren, M., Piétrac, L., and Niel, É. (2008). A supervisor implementation approach in discrete controller synthesis. In *ETFA*, 1433–1440.

Guerrouat, A. and Richter, H. (2005). A component-based specification approach for embedded systems using FDTs. In *Proceedings of the 2005 conference on Specification and verification of component-based systems*, SAVCBS '05. ACM, New York, NY, USA.

Hajjar, S., Dumitrescu, E., Niel, E., et al. (2013). Safe design method of embedded control systems : Case study. In *5èmes Journées Doctorales / Journées Nationales MACS Ecole en Modélisation, Analyse et Conduite des Systèmes dynamiques*. Strasbourg, France.

IEEE (2005). Ieee standard for property specification language (psl). *IEEE Std 1850-2005*, 1–143.

Jadot, J.Y. (2009). Ferrocots, from cable to chip. URL `http://www.eurailmag.com/mag/21.htm?page=14`.

Marchand, H., Bournai, P., LeBorgne, M., and Guernic, P.L. (2000). Synthesis of discrete-event controllers based on the signal environment. In *IN DISCRETE EVENT DYNAMIC SYSTEM: THEORY AND APPLICATIONS*, 325–346.

Ramadge, P. and Wonham, W. (1989). The control of discrete event systems. *Proceedings of the IEEE*, 77(1), 81–98. doi:10.1109/5.21072.

Roop, P., Girault, A., Sinha, R., and Goessler, G. (2009). Specification enforcing refinement for convertibility verification. In *Proceedings of the 2009 Ninth International Conference on Application of Concurrency to System Design*, ACSD '09, 148–157. IEEE Computer Society.

Xie, F., Yang, G., and Song, X. (2007). Component-based hardware/software co-verification for building trustworthy embedded systems. *Journal of Systems and Software*, 80(5), 643–654.