A Separation Principle for Embedded System Interfacing

Lucian M. Patcas, Mark Lawford, and Tom Maibaum

Department of Computing and Software
McMaster University, Hamilton, Ontario, Canada L8S 4K1
{patcaslm,lawford,maibaum}@mcmaster.ca

Abstract. In designing systems, engineers decompose the problem into smaller, more manageable tasks. A classic example of this is the *separation principle* from control systems which allows one to decompose the design of an optimal feedback control system into two independent tasks by designing (a) an observer, and (b) a controller. We investigate an analogous result for embedded system interfacing that will allow separation of the design of the input and output hardware interfaces while still guaranteeing the ability of the software to meet the system requirements. We define the notions of observability (controllability) of the system requirements with respect to the input (output) interface. We show that for a system that can be modeled by a functional four-variable model, observability and controllability allow for the separation of the design of the input and output interfaces. We also show that this separation is not always possible for systems that need the general, relational four-variable model. By strengthening either observability or controllability, we restrict the choice of input or output interfaces, but ensure separability of their designs.

1 Introduction

In designing systems, engineers like to decompose system design into smaller, more manageable tasks. A classic example of this is a conjecture by Kalman [7] that became known as the "separation principle" or "separation theorem" for linear control systems which states that one can decompose the physical realization of a state feedback controller into two stages: (a) an observer that computes a "best approximation" of the physical plant's state based upon the observations of the physical plant's outputs (monitored quantities), and (b) computation of the control signals to the plant (the control outputs) assuming access to perfect state information from the plant. When the actual plant's state is replaced in (b) by the approximation computed in (a), it can be shown that an optimal control results [5].

For reasons of flexibility and cost, the designs produced by control engineers are usually implemented as software-controlled embedded systems. A general view of an embedded system based upon [13] is depicted by the inner loop of Fig. 1. Based on the measured values of plant parameters obtained from the sensors, the software controller commands the actuators with the purpose of maintaining certain properties in the plant. Parnas and Madey's four-variable model [11] (outside square of Fig. 1) helps to clarify the behaviour of, and the boundaries between, the plant, sensors, actuators, and control software. The model has been used successfully for the past five decades in the development of safety-critical embedded systems in various industries [14,3,8,15]. The four-variable model was also used extensively in the *Requirements Engineering Handbook* [9] that was put together at the request of the U.S. Federal Aviation Administration.

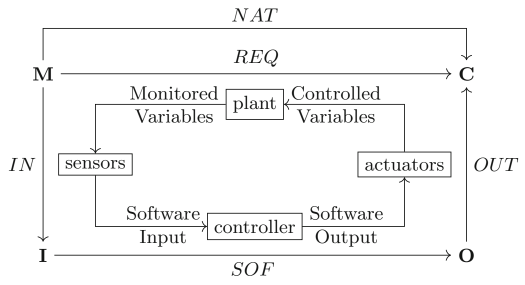


Fig. 1. The four-variable model

In the four-variable model there are four types of "variables" (hence the name): *monitored variables* (physical parameters of interest in the plant such as temperatures, voltages, aileron angle in a plane wing etc.); *controlled variables* (the physical parameters the system attempts to control); *input variables* (the digital representations of the monitored variables available to the software); and *output variables* (the variables set by the software in order to modify controlled variables). The sets of the possible values of the monitored and controlled variables are denoted by **M** and **C**, respectively; the sets of the possible values of the input and output variables are denoted by **I** and **O**, respectively. The *system requirements* *REQ* relate values of monitored variables to values of controlled variables. The environmental constraints on the system are described by the relation *NAT* (from "nature"), which restricts the possible values of the monitored and controlled variables. An environmental constraint might be, for instance, the maximum rate of climb of an aircraft in the case of an avionics system. The possible *system implementations* (system designs) are modelled by a sequential composition of *IN*, *SOF*, and *OUT*. Here, *IN* models the *input hardware interface* (sensors and analog-to-digital converters) and relates values of monitored variables to values of input variables. The *output hardware interface* (digital-to-analog converters and actuators) is modelled by *OUT*, which relates values of output variables to values of controlled variables. Relating values of input variables to values of output variables is *SOF*, which models the *control software*.

To account for the inaccuracies introduced by the hardware interfaces, IN and OUT are in general relations, not functions. For example, assume that IN models an A/D converter that converts analog voltages in the range 0–5V with an accuracy of ± 0.5V; then, for an actual monitored voltage of 2.5V, the value of the corresponding input variable in the software can be any of the digital representations that correspond to 2V, 2.5V, and 3V. A typical engineering practice is to allow tolerances on requirements (i.e., more outputs acceptable for the same input), in which case REQ is a relation as well [8]. If we want to capture all the possible implementations of the control software (i.e., the software requirements), then SOF will typically have to be a relation. An actual implementation of SOF is a deterministic program that runs on a computer and can be modeled by a function.

The relations NAT and REQ are described by application domain experts and control engineers. The system designers allocate the system requirements between hardware and software, and describe IN and OUT. The software engineers must determine SOF and verify whether it is acceptable with respect to NAT, REQ, IN, and OUT. A difficult part in designing a system is to come up with the right triple IN, SOF, and OUT such that their integration produces an acceptable system design. For complex projects that require numerous subcontractors, communication and agreement between the various teams tend to be challenging, especially when the teams are large and geographically dispersed. Being able to design the input and output interfaces separately would:

– help designers manage with system design complexity;
– reduce the interaction required between the various teams;
– allow changes to the input (output) interface without requiring changes to the output (input) interface, an idea similar to Parnas' information hiding principle [10] that prevents local changes from propagating throughout other parts of the system.

At the same time, it would also be highly desirable for the pair of input/output interfacing to not prevent acceptable software implementations from being possible. The control software must be able to observe specific changes in the monitored variables via the input interface and react to these changes by modifying the values of the controlled variables via the output interface, as specified in the requirements. Thus in our attempt at a separation principle for embedded systems interfacing, IN plays a role similar to Kalman's observer and OUT plays a role similar to Kalman's controller.

To address the deficiencies of the software acceptability notion presented in [11], we proposed in [12] a new semantics for the four-variable model based on the demonic calculus of relations. Using this semantics, we formalized software acceptability and proved a necessary and sufficient condition for an acceptable software implementation to exist. In the current paper we revisit this condition in Section 3 and present it from a different angle by introducing the notions of observability and controllability of requirements with respect to the input, and, respectively, output interfaces. As it turns out, this necessary and sufficient condition has a surprising practical implication: if functions are used, the input and output interfaces can always be designed independently and an acceptable software implementation will still be possible as long as the observability and controllability conditions both hold; in the relational case, however, the input and output hardware interfaces are, in general, mutually dependent. Since relational specifications are more realistic in practice because they can model the nondeterminism induced by hardware inaccuracies and tolerances on requirements, in Section 4 we prove two stronger conditions that allow the input and output interfaces to be designed independently while still guaranteeing the ability of the software to meet the system requirements. In Section 5 we discuss some of the practical and theoretical implications of our results as well as limitations and future research directions.

2 Mathematical Preliminaries

The mathematics presented in this section will be applied to the four-variable model in the subsequent sections of the paper. We take a semantic view and consider that relations are models of specifications as well as of actual implementations.

2.1 Relations and Covers

A relation R from a set A to a set B is a subset of the cartesian product $A \times B$. In other words, R is a subset of the set of all ordered pairs (a, b), where $a \in A$ and $b \in B$. Some operations involving a relation $R \subseteq A \times B$ are:

- *domain* of R: $\mathsf{dom}\,(R) = \{a \in A \mid \exists b \in B.\ (a, b) \in R\}$;
- *range* of R: $\mathsf{ran}\,(R) = \{b \in B \mid \exists a \in A.\ (a, b) \in R\}$;
- *converse* of R: $R^{\smile} = \{(b, a) \in B \times A \mid (a, b) \in R\}$;
- *image set* of $a \in A$ under R: $R(a) = \{b \in B \mid (a, b) \in R\}$.

The image set of an element in the domain of a relation denotes the inaccuracy or tolerance acceptable for that input.

A relation $R \subseteq A \times B$ is *univalent* if it maps every element in its domain to exactly one element in its range. Univalent relations also go by the name *functional relations* or *partial functions*. Relation R is *total* if and only if $\mathsf{dom}\,(R) = A$. The relations that are both univalent and total are called *mappings* or *total functions*.

A *cover* of a set A is a family $\mathcal{C} = \{C_\alpha \subseteq A \mid \alpha \in \mathcal{I}\}$ where α is an index in some index set $\mathcal{I}$, $A = \bigcup_{\alpha \in \mathcal{I}} C_\alpha$, and the subsets C_α of A, called the *cells* of $\mathcal{C}$, are not necessarily pairwise disjoint. A particular case of a cover is the Wonham cover induced by a relation on its domain [16]. The Wonham cover induced by $R \subseteq A \times B$ on $\mathsf{dom}\,(R)$ is:

$$\mathsf{cov}\,(R) = \left\{A' \subseteq A \;\middle|\; \exists b \in \mathsf{ran}\,(R).\ A' = R^{\smile}(b)\right\}. \tag{1}$$

The cells of $\mathsf{cov}\,(R)$, indexed by $\mathsf{ran}\,(R)$, are the image sets of the elements in the range of R under the converse of R. In the sequel, when we use the word cover we will mean a Wonham cover.

2.2 Demonic Factorization of Relations

For the goals set in Section 1, we are interested in existence conditions for the dotted arrows in the commutative diagram depicted in Fig. 2. This diagram is isomorphic to the four-variable model diagram.

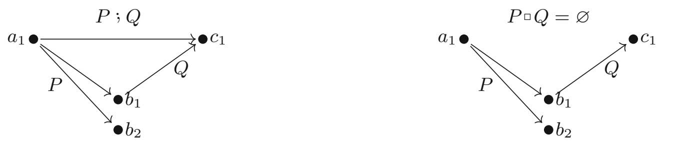
$$\begin{tikzcd}
A \arrow[rr, "R"] \arrow[d, "P"'] \arrow[rrd, dashed, "Y \sqsubseteq R \mathbin{/\!\!/} Q"] & & D \\
B \arrow[rru, dashed, "X \sqsubseteq P \mathbin{\backslash\!\!\backslash} R"'] \arrow[rr, dashed, "Z \sqsubseteq P \mathbin{\backslash\!\!\backslash} R \mathbin{/\!\!/} Q"'] & & C \arrow[u, "Q"']
\end{tikzcd}$$

Fig. 2. Demonic factorization

The *composition* of two relations $P \subseteq A \times B$ and $Q \subseteq B \times C$ is the relation

$$P \mathbin{;} Q = \{(a,c) \in A \times C \mid \exists b \in B.\ (a,b) \in P \wedge (b,c) \in Q\}\,. \tag{2}$$

The problem with this notion of composition is that specifications P and Q are allowed where some points in the range of P are not in the domain of Q. In practice, this means that an implementation of $P \mathbin{;} Q$ will not always produce a result when expected to. Consider, for instance, the relations $P = \{(a_1, b_1), (a_1, b_2)\}$ and $Q = \{(b_1, c_1)\}$, depicted in Fig. 3. In this example, $P \mathbin{;} Q$ allows the dead end (a_1, b_2) because a_1 can still reach c_1 via b_1. Semantics that allow such behaviours are called *angelic*. In angelic semantics, specifications that allow "bad" behaviours for some inputs are permitted as long as they also allow "good" behaviours for those inputs. In contrast, a *demonic semantics* rejects any specification that allows "bad" behaviours. Considering that many embedded systems are used in safety-critical applications, it is always wise to plan for the worst, hence we find a demonic semantics more adequate.



(a) Angelic composition (b) Demonic composition

Fig. 3. Composition of relations

The *demonic composition* of P with Q is the relation

$$P \mathbin{\square} Q = \{(a,c) \in A \times C \mid (a,c) \in P \mathbin{;} Q \wedge P(a) \subseteq \mathsf{dom}\,(Q)\}\,. \tag{3}$$

As can be seen in Fig. 3, $P \mathbin{\square} Q$ is empty for those inputs for which there is a chance of not producing expected results, thus it is our choice of sequential composition in the four-variable model. Demonic and angelic compositions are the same when P is univalent or Q is total.

The following subrelation of a relation $P \subseteq A \times B$ is obtained by restricting the domain of P to the domain of another relation $R \subseteq A \times C$:

$$P\big|_{\mathsf{dom}(R)} = \{(a,b) \in P \mid a \in \mathsf{dom}\,(R)\}\,. \tag{4}$$

This construction is helpful when working with partial relations. The rationale for allowing partial relations is that, in practice, in the early stages of system development it is more likely that incomplete specifications are produced rather than total specifications. More detail is added as the system becomes better understood and, eventually, the specifications will cover all the possible cases that can arise. Before getting to that point, however, many useful analyses can be performed, such as the implementability checks described in Sections 3 and 4.

A relation $P \subseteq A \times B$ is a *demonic refinement* of a relation $R \subseteq A \times B$, written $P \sqsubseteq R$, if and only if $\mathsf{dom}\,(R) \subseteq \mathsf{dom}\,(P)$ and $P\big|_{\mathsf{dom}(R)} \subseteq R$. Demonic refinement is also known as total correctness in [2,6]. The intuition for demonic refinement is as follows:

- for every input in the domain of R, P must produce only outputs allowed by R (i.e., an implementation is at least as deterministic as its specification);
- for the inputs outside the domain of R, P is allowed to produce any output or no output at all.

Demonic refinement is a partial order on relations. As an example, consider the relations in Fig. 4: $R = \{(a_1,b_1),(a_1,b_2),(a_2,b_2)\}$, $P = \{(a_1,b_1),(a_2,b_2),$ $(a_3,b_2),(a_3,b_3)\}$, and $Q = \{(a_1,b_1),(a_2,b_1),(a_3,b_2),(a_3,b_3)\}$. Here, P refines R, but Q does not refine R because $(a_2,b_1) \notin R$.

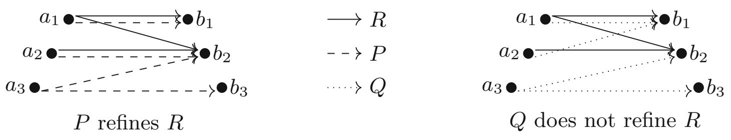


Fig. 4. Examples of demonic refinement

Demonic composition and demonic refinement induce two residuation operations, the demonic left and right residuals. If composition is seen as a multiplicative operation, then the residuation operations play the role of division and their results are quotients. The demonic left and right residuals are useful when a relation is refined by a demonic composition of two relations and one of these relations is not known, as in triangles $\triangle A,B,D$ and $\triangle A,C,D$ in Fig. 2. The *demonic left residual* of R by Q, denoted $R \mathbin{/\!\!/} Q$, is defined as the largest solution, with respect to $\sqsubseteq$, of the inequation $Y \,\square\, Q \sqsubseteq R$, where Y is the unknown:

$$Y \,\square\, Q \sqsubseteq R \Leftrightarrow Y \sqsubseteq R \mathbin{/\!\!/} Q\,. \tag{5}$$

A solution Y, called a *demonic left factor* of R through Q, does not always exist. In [12], we proved the following necessary and sufficient condition for the existence of a demonic left factor:

$$(\exists Y.\ Y \,\square\, Q \sqsubseteq R) \Leftrightarrow \forall a \in \mathsf{dom}\,(R).\ \exists c \in \mathsf{dom}\,(Q).\ Q(c) \subseteq R(a)\,. \tag{6}$$

According to (5), the demonic left residual $R \mathbin{/\!\!/} Q$ is defined only when a demonic left factor exists. If $R \mathbin{/\!\!/} Q$ is defined, then its value is:

$$R \mathbin{/\!\!/} Q \coloneqq \{(a,c) \in A \times C \mid c \in \mathsf{dom}\,(Q) \wedge Q(c) \subseteq R(a)\}\,. \tag{7}$$

The symbol $\coloneqq$, called "venturi tube" [6], has the following meaning: for any two expressions ϕ and ψ, if $\phi \coloneqq \psi$, then ψ is defined and equal to ϕ if and only if ϕ is defined.

Similarly to the demonic left residual, the *demonic right residual* of R by P, denoted $P \mathbin{\backslash\!\!\backslash} R$, is defined as the largest solution, with respect to $\sqsubseteq$, of the inequation $P \,\square\, X \sqsubseteq R$, where X is the unknown:

$$P \,\square\, X \sqsubseteq R \Leftrightarrow X \sqsubseteq P \mathbin{\backslash\!\!\backslash} R\,. \tag{8}$$

A solution X, called a *demonic right factor* of R through P, does not always exist. Therefore, by (8), the demonic right residual $P \mathbin{\backslash\!\!\backslash} R$ is not always defined. We proved in [12] that the following condition is necessary and sufficient for the existence of a demonic right factor and for the definedness of the demonic right residual:

$$\begin{aligned}&(\exists X.\ P \,\square\, X \sqsubseteq R) \Leftrightarrow \\ &\mathsf{dom}\,(R) \subseteq \mathsf{dom}\,(P) \wedge \forall b \in \mathsf{ran}\left(P|_{\mathsf{dom}(R)}\right).\ \exists d \in D.\ \left(P|_{\mathsf{dom}(R)}\right)^{\smile}(b) \subseteq R^{\smile}(d)\,.\end{aligned} \tag{9}$$

If $P \mathbin{\backslash\!\!\backslash} R$ is defined, then its value is:

$$P \mathbin{\backslash\!\!\backslash} R \coloneqq \left\{(b,d) \in B \times D \;\middle|\; b \in \mathsf{ran}\left(P|_{\mathsf{dom}(R)}\right) \wedge \left(P|_{\mathsf{dom}(R)}\right)^{\smile}(b) \subseteq R^{\smile}(d)\right\}. \tag{10}$$

The demonic left and right residuals are also useful when we wish to decompose a relation into a demonic composition of three relations and the relation in the middle is not known. This situation is depicted in Fig. 2, where we are interested in solving the inequality $P \,\square\, Z \,\square\, Q \sqsubseteq R$ for Z. A solution Z, which we call a *demonic mid factor* of R through P and Q, does not always exist. In [12] we showed that:

$$\begin{aligned}&(\exists Z.\ P \,\square\, Z \,\square\, Q \sqsubseteq R) \Leftrightarrow \\ &\mathsf{dom}\,(R) \subseteq \mathsf{dom}\,(P) \wedge \forall b \in \mathsf{ran}\left(P|_{\mathsf{dom}(R)}\right).\ \exists c \in \mathsf{dom}\,(Q). \\ &\qquad Q(c) \subseteq \left\{d \in D \;\middle|\; \left(P|_{\mathsf{dom}(R)}\right)^{\smile}(b) \subseteq R^{\smile}(d)\right\}.\end{aligned} \tag{11}$$

We also proved that any demonic mid factor is a demonic refinement of the residual $P \mathbin{\backslash\!\!\backslash} R \mathbin{/\!\!/} Q$, which we call the *demonic mid residual* of R by P and Q. In other words, this residual is the largest solution, with respect to $\sqsubseteq$, of the inequality $P \,\square\, Z \,\square\, Q \sqsubseteq R$:

$$P \,\square\, Z \,\square\, Q \sqsubseteq R \Leftrightarrow Z \sqsubseteq P \mathbin{\backslash\!\!\backslash} R \mathbin{/\!\!/} Q\,. \tag{12}$$

If a demonic mid factor exists, then the value of $P \mathbin{\backslash\!\!\backslash} R \mathbin{/\!\!/} Q$ is well defined and is given by:

$$P \mathbin{\backslash\!\!\backslash} R \mathbin{/\!\!/} Q \;\widehat{=}\; \left\{(b,c) \in B \times C \;\middle|\; b \in \mathsf{ran}\left(P|_{\mathsf{dom}(R)}\right) \wedge c \in \mathsf{dom}\,(Q) \wedge \right.$$
$$\left. Q(c) \subseteq \left\{d \in D \;\middle|\; \left(P|_{\mathsf{dom}(R)}\right)^{\smile}(b) \subseteq R^{\smile}(d)\right\}\right\}\,. \tag{13}$$

More details on the demonic calculus of relations can be found in [4,1,2,6,12].

3 Implementability

In this section we ask the question of implementability of system requirements: is an acceptable implementation of the system requirements possible given a particular choice of hardware interfacing between the system and the physical environment? We present necessary and sufficient implementability conditions in both the functional and relational cases of the four-variable model. For the reasons mentioned in the introduction, we would like to be able to design the input and output interfaces independently of each other, while ensuring that an acceptable implementation is still possible. As it turns out, this separation is always possible in the functional setting, but not always when relational specifications are used.

To not overcomplicate the presentation, in this paper we do not use the relation NAT explicitly; instead, we assume that the system requirements specify only physically meaningful outputs for the inputs that are possible from the environment. More details about how NAT affects implementability can be found in [12].

We now return to the question of implementability of system requirements and give the following definition for implementability.

Definition 1. *System requirements REQ are implementable if an input interface IN, an output interface OUT and software SOF exist such that* $IN \,\square\, SOF \,\square\, OUT \sqsubseteq REQ$.

In Definition 1, a system implementation is given by the demonic composition of *IN*, *SOF*, and *OUT*. As explained in Section 2.2, the demonic composition ensures that there are no dead ends when integrating *IN*, *SOF*, and *OUT*. As a satisfaction criterion, we use the demonic refinement of relations, which ensures that for every input in the domain of the requirements an implementation will produce only results allowed by the requirements. The demonic refinement allows arbitrary system behaviour for the inputs outside the domain of the requirements, but this should present no danger as it is assumed that for a final product hazard analyzes have been conducted and all the inputs that could lead to hazardous system behaviour have been added to the domain of REQ as additional safety requirements. We call *acceptable* a system implementation $IN \mathbin{\square} SOF \mathbin{\square} OUT$ such that $IN \mathbin{\square} SOF \mathbin{\square} OUT \sqsubseteq REQ$. Definition 1 implies that the system requirements are implementable only if an acceptable system implementation exists. A software implementation is *acceptable* if and only if it is part of an acceptable system implementation. Therefore, the implementability of system requirements reduces to the existence of an acceptable software implementation, which is relative to the choices made by the system designers for the input and output hardware.

The question now is when does an acceptable software implementation exist? The software must be able to observe specific changes in the monitored variables via the input interface and react to these changes by modifying the values of the controlled variables via the output interface, as specified in the requirements. We introduce the notions of observability and controllability of system requirements with respect to the input and, respectively, output hardware interfaces.

Definition 2. *System requirements REQ are* observable *with respect to an input interface IN if there exists a demonic right factor of REQ through IN.*

For system requirements REQ to be observable, Definition 2 requires that there exists a relation $X \subseteq \mathbf{I} \times \mathbf{C}$ such that $IN \mathbin{\square} X \sqsubseteq REQ$. Observability is a necessary condition for implementability since if $IN \mathbin{\square} SOF \mathbin{\square} OUT \sqsubseteq REQ$ we can take $X = SOF \mathbin{\square} OUT$. Intuitively, observability says that in the worst case IN always retains at least as much information about the monitored variables as REQ.

Definition 3. *System requirements REQ are* controllable *with respect to an output interface OUT if there exists a demonic left factor of REQ through OUT.*

For system requirements REQ to be controllable, Definition 3 requires that there exists a relation $Y \subseteq \mathbf{M} \times \mathbf{O}$ such that $Y \mathbin{\square} OUT \sqsubseteq REQ$. Clearly controllability is also necessary for implementability since if $IN \mathbin{\square} SOF \mathbin{\square} OUT \sqsubseteq REQ$ we can always take $Y = IN \mathbin{\square} SOF$. The intuition for controllability is that in the worst case OUT must be at least as precise as REQ.

In the remainder of the section, we will discuss how observability and controllability affect implementability of system requirements in both the functional and relational cases of the four-variable model.

3.1 Functional Case

Here we assume the extreme case where the specifications in the four-variable model are all total functions.

Proposition 1. *System requirements REQ are observable with respect to an input interface IN if and only if* $\forall M' \in \mathsf{cov}(IN).\ \exists M'' \in \mathsf{cov}(REQ).\ M' \subseteq M''$.

Proof. By (1) and specializing (9) to total functions. □

Proposition 2. *System requirements REQ are controllable with respect to an output interface OUT if and only if* $\mathsf{ran}\,(REQ) \subseteq \mathsf{ran}\,(OUT)$.

Proof. By specializing (6) to total functions. □

Proposition 3. *System requirements REQ are implementable with respect to an input interface IN and an output interface OUT if and only if REQ is observable with respect to IN and controllable with respect to OUT.*

Proof. By specializing (11) to total functions. □

Because observability is defined only in terms of *REQ* and *IN*, and controllability only in terms of *REQ* and *OUT*, a corollary of Prop. 3 is that for an acceptable *SOF* to exist, *IN* and *OUT* are always separable. The practical implication is that the input and output interfaces of a system modeled using a functional four-variable model can always be designed independently and an acceptable software implementation is guaranteed to exist.

3.2 Relational Case

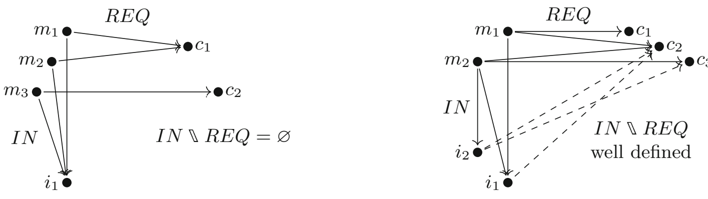
We now consider the most general case where the specifications in the four-variable model are partial relations.

Proposition 4. *System requirements REQ are observable with respect to an input interface IN if and only if the following conditions are both satisfied:*

(i) $\mathsf{dom}\,(REQ) \subseteq \mathsf{dom}\,(IN)$;
(ii) $\forall M' \in \mathsf{cov}\left(IN|_{\mathsf{dom}(REQ)}\right).\ \exists M'' \in \mathsf{cov}\,(REQ).\ M' \subseteq M''$.

Proof. Follows from (1) and (9). □

Proposition 4(i) requires an input interface to "see" every input for which the requirements specify system behaviour. Proposition 4(ii), also known as refinement of covers in mathematical topology, requires the accuracy of the input interface to be the same or of finer granularity than what the requirements imply. For example, in Fig. 5a, $\mathsf{cov}\left(IN|_{\mathsf{dom}(REQ)}\right) = \{\{m_1, m_2, m_3\}\}$ and $\mathsf{cov}\,(REQ) = \{\{m_1, m_2\}, \{m_3\}\}$. The cell $IN^{\smile}(i_1) = \{m_1, m_2, m_3\}$ in $\mathsf{cov}\left(IN|_{\mathsf{dom}(REQ)}\right)$ corresponds to i_1 and represents the accuracy with which *IN* produces i_1; in other words, the software is not able to distinguish between m_1, m_2, or m_3 when it receives the input i_1. The requirements in this example, on the other hand, require the system to make a distinction in how it treats m_3 compared to m_1 and m_2, reflected by the two distinct cells $REQ^{\smile}(c_2) = \{m_3\}$ and, respectively, $REQ^{\smile}(c_1) = \{m_1, m_2\}$ in $\mathsf{cov}\,(REQ)$. The software will not be able to make this distinction because the cell $\{m_1, m_2, m_3\}$ in $\mathsf{cov}\left(IN|_{\mathsf{dom}(REQ)}\right)$ is not contained in any of the cells of $\mathsf{cov}\,(REQ)$. Consequently, the accuracy of *IN* is coarser than required and *REQ* is not observable with respect to *IN*. In the example depicted in Fig. 5b, $\mathsf{cov}\left(IN|_{\mathsf{dom}(REQ)}\right) = \{\{m_1, m_2\}, \{m_2\}\}$ and $\mathsf{cov}\,(REQ) = \{\{m_1\}, \{m_1, m_2\}, \{m_2\}\}$ satisfy Prop. 4(ii). Because $\mathsf{dom}\,(REQ) = \mathsf{dom}\,(IN)$, Prop. 4(i) is also satisfied, hence *REQ* is observable with respect to *IN*, ensuring that there is a way to relate the software inputs to values of controlled variables via a demonic right factor of *REQ* through *IN*. Note that $IN \mathbin{\backslash\!\!\backslash} REQ$ is the largest, with respect to $\sqsubseteq$, such factor (i.e., the least restrictive specification).



REQ
m_1
m_2
m_3
c_1
c_2
IN
i_1
$IN \mathbin{\backslash\!\!\backslash} REQ = \varnothing$
REQ
m_1
m_2
c_1
c_2
c_3
IN
i_2
i_1
$IN \mathbin{\backslash\!\!\backslash} REQ$
well defined

(a) *REQ* is not observable (b) *REQ* is observable

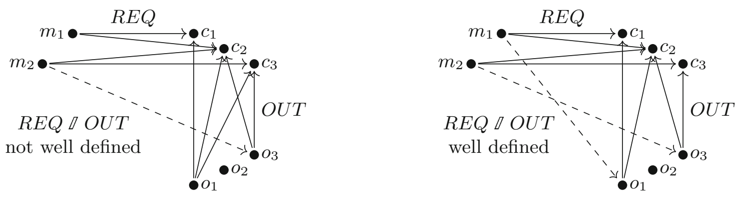
Fig. 5. Observability

Proposition 5. *System requirements REQ are controllable with respect to an output interface OUT if and only if* $\forall C' \in \mathsf{cov}\left(REQ^{\smile}\right).\ \exists C'' \in \mathsf{cov}\left(OUT^{\smile}\right).\ C'' \subseteq C'$.

Proof. Follows from (1) and (6). □

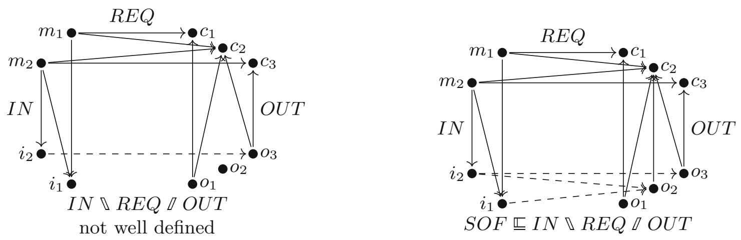
The intuition for Prop. 5 is that for the system requirements to be controllable the output hardware should allow for the same or finer control over the controlled variables than what is implied by the requirements. The cells in the covers of $REQ^{\smile}$ or $OUT^{\smile}$ are measures of the amount of control: the smaller the cell, the more precise the control. For example, in Fig. 6a the cell $REQ(m_1) = \{c_1, c_2\}$ in $\mathsf{cov}\left(REQ^{\smile}\right)$ does not contain any of the cells of $\mathsf{cov}\left(OUT^{\smile}\right)$. As such, *OUT* does not have the right amount of control over the controlled variables and *REQ* is not controllable with respect to *OUT*. Figure 6b depicts an example where there is a way to relate the monitored values to software outputs via a demonic left factor of *REQ* through *OUT* and, consequently, *REQ* is controllable. Note that $REQ \mathbin{/\!\!/} OUT$ is the largest, with respect to $\sqsubseteq$, such factor (i.e., the least restrictive specification).

In contrast to the functional case, in the relational case observability and controllability are not sufficient for implementability. A counterexample to the sufficiency of their conjunction is given in Fig. 7a, which combines the examples from Figs. 5b and 6b. In this example, *REQ* is observable and controllable even though there is no acceptable software. By (12), any acceptable software implementation is a demonic refinement of $IN \mathbin{\backslash\!\!\backslash} REQ \mathbin{/\!\!/} OUT$, which is not well defined here. The reason for this is that i_1 cannot be connected with either



(a) *REQ* is not controllable (b) *REQ* is controllable

Fig. 6. Controllability



(a) *REQ* is not implementable (b) *REQ* is implementable

Fig. 7. Implementability

o_1 or o_2 without breaking demonic refinement. For example, if we connect i_1 with o_1, then m_2 will be connected with c_1 via $IN \,\square\, SOF \,\square\, OUT$, something not allowed by *REQ*. If we extend *OUT* with the pair (o_2, c_2) as in Fig. 7b, then $IN \setminus\!\!\setminus REQ /\!\!/ OUT$ becomes well defined, hence an acceptable *SOF* is possible. The demonic mid residual $IN \setminus\!\!\setminus REQ /\!\!/ OUT$ is the least restrictive specification for acceptable software.

Proposition 6. *System requirements REQ are implementable with respect to an input interface IN and an output interface OUT if and only if the following two conditions are both satisfied:*

(i) $\mathsf{dom}\,(REQ) \subseteq \mathsf{dom}\,(IN)$;

(ii) $\forall M' \in \mathsf{cov}\left(IN\big|_{\mathsf{dom}(REQ)}\right).\ \exists C' \in \mathsf{cov}\left(OUT^{\smile}\right).\ C' \subseteq \bigcap_{m \in M'} REQ(m)$.

Proof. Follows from (1) and (11). □

The conditions in Prop. 6 imply both observability and controllability. However, the requirements are implementable if and only if a certain balance exists between observability and controllability. In Fig. 7b, *REQ* is implementable because if we consider the cell $IN^{\smile}(i_1) = \{m_1, m_2\}$ in $\mathsf{cov}\left(IN\big|_{\mathsf{dom}(REQ)}\right)$, then there is the cell $OUT^{\smile}(o_2) = \{c_2\} = REQ(m_1) \cap REQ(m_2)$ in $\mathsf{cov}\left(OUT^{\smile}\right)$; similarly, for $IN^{\smile}(i_2) = \{m_2\}$ in $\mathsf{cov}\left(IN\big|_{\mathsf{dom}(REQ)}\right)$, there is $OUT^{\smile}(o_3) = \{c_2, c_3\} = REQ(m_2)$ in $\mathsf{cov}\left(OUT^{\smile}\right)$, hence Prop. 6(ii) is satisfied.

As can be seen in Prop. 6(ii), *IN* and *OUT* are coupled. In practice, this means that for the requirements to be implementable, the input and output hardware cannot be, in general, designed independently of each other.

4 Separability

In this section, we present two stronger implementability conditions for the relational setting that allow the input and output hardware to be designed independently of each other.

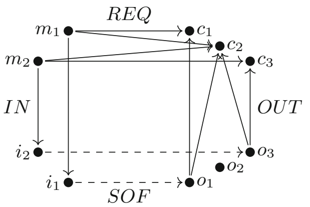
We obtain the first stronger implementability condition by strengthening controllability as follows.

Proposition 7. *System requirements REQ are implementable with respect to an input interface IN and an output interface OUT if the following two conditions are both satisfied:*

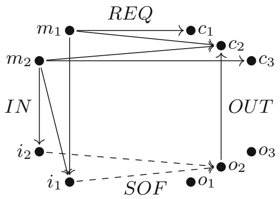
(i) REQ is observable with respect to IN;
(ii) $\forall M' \in \mathsf{cov}\,(REQ).\ \exists C' \in \mathsf{cov}\left(OUT^{\smile}\right).\ C' \subseteq \bigcap_{m \in M'} REQ(m)$.

Proof. To prove the implementability of REQ we have to show that Prop. 6 is satisfied. Proposition 6(i) follows easily from Prop. 7(i). Also from Prop. 7(i), we have that for any $M' \in \mathsf{cov}\left(IN\big|_{\mathsf{dom}(REQ)}\right)$ there is a $M'' \in \mathsf{cov}\,(REQ)$ such that $M' \subseteq M''$. If we substitute M'' for M' in Prop. 7(ii), we get that there exists a $C' \in \mathsf{cov}\left(OUT^{\smile}\right)$ such that $C' \subseteq \bigcap_{m \in M''} REQ(m)$. Because $M' \subseteq M''$, we also have that $C' \subseteq \bigcap_{m \in M'} REQ(m)$. In conclusion, we have proved that for any $M' \in \mathsf{cov}\left(IN\big|_{\mathsf{dom}(REQ)}\right)$ there is a $C' \in \mathsf{cov}\left(OUT^{\smile}\right)$ such that $C' \subseteq \bigcap_{m \in M'} REQ(m)$, which is exactly Prop. 6(ii). □

We call a relation REQ that satisfies Prop. 7(ii) *strongly controllable* with respect to OUT. An example of strongly controllable requirements is in Fig. 7b. Strong controllability is not necessary for implementability, as shown in Fig. 8a. Here, the requirements are implementable and, consequently, controllable with respect to OUT, although they are not strongly controllable. As such, strong controllability reduces the number of output hardware choices when compared with controllability. On the other hand, strong controllability ensures that IN and OUT can be chosen independently of each other as long as they satisfy their respective constraints in Prop. 7.



(a) REQ is implementable, but not strongly controllable



(b) REQ is implementable, but not strongly observable

Fig. 8. Strong observability and controllability not necessary for implementability

In the second stronger implementability condition, we strengthen observability as follows.

Proposition 8. *System requirements REQ are implementable with respect to an input interface IN and an output interface OUT if the following conditions are all satisfied:*

(i) $\mathsf{dom}\left(REQ\right) \subseteq \mathsf{dom}\left(IN\right)$;
(ii) $\forall M' \in \mathsf{cov}\left(IN|_{\mathsf{dom}(REQ)}\right).\ \exists C' \in \mathsf{cov}\left(REQ^{\smile}\right).\ M' \subseteq \bigcap_{c \in C'} REQ^{\smile}(c)$;
(iii) REQ is controllable with respect to OUT.

Proof. Similar to the proof for Prop. 7. □

We call *REQ strongly observable* with respect to *IN* if *REQ* and *IN* satisfy Props. 8(i) and 8(ii). An example of strongly observable requirements is in Fig. 8a. Strong observability is not necessary for implementability (Fig. 8b). In this example, the requirements are implementable without Prop. 8(ii) being satisfied. As such, strong observability restricts the acceptable choices of input hardware compared with observability, but at the same time it allows the separation of *IN* and *OUT* as long as they satisfy the constraints of Prop. 8.

5 Discussion

In this paper, we presented one necessary and sufficient (Prop. 6) and two sufficient (Props. 7 and 8) implementability conditions that allow the system designers to choose a pair of input and output hardware interfaces such that an acceptable software implementation is guaranteed to exist. Implementability does not imply that implementing the SOF relation is practical. Nevertheless, it gives software engineers the confidence that their efforts are not destined to fail from the beginning. If implementability is not satisfied, then no acceptable implementation will be possible.

From a system development perspective, an important question is which implementability condition to use and when. If separating *IN* and *OUT* at design time is important, then one of the stronger implementability conditions should be used as follows:

– if the input hardware is more difficult to design than the output hardware, then it is desirable to have as many options as possible for the input hardware. In such cases, Prop. 7 is more suitable because the implied strong controllability limits only the choices of output hardware without overly-restricting the input hardware. If for observability the necessary and sufficient condition of Prop. 4 is used, then this will allow the widest possible range of acceptable input hardware;
– similarly, Prop. 8 together with Prop. 5 should be used if having more design options for the output hardware is more important than for the input hardware.

If the system designers need as many acceptable options as possible for both the input and output interfaces, and separability of *IN* and *OUT* is not as important, then the necessary and sufficient implementability conditions in Prop. 6 should be used.

The stronger implementability conditions in Props. 7 and 8 can be viewed as a "separation principle" for embedded systems interfacing similar to the well known separation principle for linear control systems design [7]. The analogy is not perfect, however. An observer in the control engineering sense would be constructed in the four-variable model as a simulation of a linear system inside *SOF*. The relation *IN* represents the input hardware that obtains the samples that would be used as input to the observer simulation. Similarly, a state feedback controller in the control engineering sense would be computed as a matrix multiplication inside *SOF*, the results of which would then be sent to the physical plant via the output hardware represented by *OUT*. Also, in control engineering observability and controllability of a plant are sufficient for separability of observers and controllers, while in the relational four-variable model either observability or controllability of *REQ* needs to be strengthened in order for the designs of the input and output interfaces to be separable.

The results presented in this paper are very general. The relations *REQ*, *IN*, *OUT*, and *SOF* model input-output behaviours without internal states. Also, we did not assume any structure on the sets $\mathbf{M}$, $\mathbf{C}$, $\mathbf{I}$, and $\mathbf{O}$. Because of this generality, our implementability conditions do not explicitly consider constraints that a practical implementation has to deal with, such as timing. In our current formalization, the sets $\mathbf{M}$, $\mathbf{C}$, $\mathbf{I}$, and $\mathbf{O}$ contain all the possible values for every, respectively, monitored, controlled, input, and output variable. Time can be added explicitly to the four-variable model by treating the elements of $\mathbf{M}$, $\mathbf{C}$, $\mathbf{I}$, and $\mathbf{O}$ as functions of time [11,8]. A useful research direction would be to specialize our implementability conditions to their timed versions.

The results also have applicability beyond embedded systems. They can be applied to essentially any system that can be modeled using a commutative diagram similar to the one of the four-variable model (Figs. 1 and 2). Such commutative diagrams also appear in stepwise refinement techniques where mappings between behaviours at different levels of abstraction are rather frequent.

To be useful in practice, our implementability checks need to be supported by tools. For a completely automated check, SMT solving may be a fruitful direction, although many SMT solvers do not cope well with formulas that have existential quantifiers within the scope of universal quantifiers. Another approach would be to develop heuristic algorithms for the problem at hand. When SMT solving and heuristics do not work, or in the case of very large or infinite relations, verifying implementability will still be possible in an higher-order proof assistant such as Coq, Isabelle, PVS etc., paying the price of having to do tedious and, more than often, hard proofs.

We have formalized and checked the mathematics presented in the paper with the proof assistant Coq. The files are available at `www.cas.mcmaster.ca/~patcaslm/papers/2014-iFM/coq`.

References

1. Brink, C., Kahl, W., Schmidt, G. (eds.): Relational Methods in Computer Science. Advances in Computing. Springer (1997)
2. Desharnais, J., Mili, A., Nguyen, T.: Refinement and Demonic Semantics. In: Brink, et al. (eds.) [1], ch. 11, pp. 166–183 (1997)
3. Faulk, S., Finneran, J., Kirby, J., Shash, S., Sutton, J.: Experience applying the CoRE method to the Lockhead C-130J software requirements. In: Ninth Annual Conference on Computer Assurance, Gaithersburg, Maryland (June 1994)
4. Frappier, M.: A Relational Basis for Program Construction by Parts. Ph.D. thesis, Computer Science Department, University of Ottawa (1995)
5. Joseph, D.P., Tou, T.J.: On linear control theory. Transactions of the American Institute of Electrical Engineers. Part II: Applications and Industry 80(4), 193–196 (1961)
6. Kahl, W.: Refinement and development of programs from relational specifications. Electronic Notes in Theoretical Computer Science (ENTCS) 44(3), 51–93 (2003)
7. Kalman, R.E.: Contributions to the theory of optimal control. Bol. Soc. Mat. Mexicana 5(2), 102–119 (1960)
8. Lawford, M., McDougall, J., Froebel, P., Moum, G.: Practical application of functional and relational methods for the specification and verification of safety critical software. In: Rus, T. (ed.) AMAST 2000. LNCS, vol. 1816, pp. 73–88. Springer, Heidelberg (2000)
9. Lempia, D.L., Miller, S.P.: Requirements engineering management handbook. Tech. Rep. DOT/FAA/AR-08/32, U.S. Department of Transportation, Federal Aviation Administration (June 2009)
10. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Communications of the ACM 15(12), 1053–1058 (1972)
11. Parnas, D.L., Madey, J.: Functional documents for computer systems. Science of Computer Programming 25(1), 41–61 (1995)
12. Patcas, L.M., Lawford, M., Maibaum, T.: From system requirements to software requirements in the four-variable model. In: Schneider, S., Treharne, H., Margaria, T., Padberg, J., Taentzer, G. (eds.) Proceedings of the Automated Verification of Critical Systems (AVoCS 2013). Electronic Communications of the EASST, vol. 66 (2014)
13. Thompson, J., Heimdahl, M., Miller, S.P.: Specification-based prototyping for embedded systems. In: Nierstrasz, O., Lemoine, M. (eds.) ESEC/FSE 1999. LNCS, vol. 1687, pp. 163–179. Springer, Heidelberg (1999)
14. Van Schouwen, A.: The A-7 requirements model: Re-examination for real-time systems and an application to monitoring systems. Tech. Rep. 90-276, Queens University, Ontario, Canada (1990)
15. Wassyng, A., Lawford, M.: Lessons learned from a successful implementation of formal methods in an industrial project. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 133–153. Springer, Heidelberg (2003)
16. Wonham, W.M.: Lecture notes on supervisory control of discrete-event systems. Systems Control Group, Department of Electrical & Computer Engineering, University of Toronto (July 2013), `http://www.control.toronto.edu/DES/`