Science of Computer Programming 111 (2015) 339–362

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Implementability of requirements in the four-variable model

Lucian M. Patcas *, Mark Lawford, Tom Maibaum

Department of Computing and Software, McMaster University, 1280 Main St. W., Hamilton, ON L8S4K1, Canada

ARTICLE INFO

Article history:
Received 8 June 2014
Received in revised form 8 March 2015
Accepted 14 May 2015
Available online 27 May 2015

Keywords:
Safety-critical
Four-variable model
Implementability of requirements
Tolerances on requirements
Demonic calculus of relations

ABSTRACT

Many safety-critical computer systems are required to monitor and control physical processes. The four-variable model, which has been used successfully in industry for almost four decades, helps to clarify the behaviors of, and the boundaries between the physical processes, input/output devices, and software. In this model, the acceptable behaviors of the software are constrained by the physical environment, system requirements, and input/output devices. If acceptable software behaviors are possible, then the system requirements are said to be implementable with respect to these constraints. The only acceptability condition proposed in the literature deems as acceptable software behaviors that can lead to undesirable system behaviors, in particular, nondeterministic system behaviors that for the same input sometimes do not produce any results and some other times produce expected results. In this sense, the acceptability condition can be seen as angelic. In this paper we strengthen the acceptability condition using the demonic calculus of relations such that no undesirable system or software behaviors are allowed and prove a necessary and sufficient implementability condition for the system requirements. As a byproduct, we also obtain a mathematical characterization of the least restrictive software specification, which, for all intents and purposes, can play the role of the software requirements.

1. Introduction

Many safety-critical systems in application domains such as aerospace, automotive, medical devices, or nuclear power generation are required to monitor and control physical processes. An example is the shutdown system of a nuclear reactor which monitors the temperature and pressure inside the reactor and commands the reactor to enter a shutdown state whenever abnormal temperature and pressure values have been detected. Such systems are usually implemented using digital computers that are embedded into the larger system of the application and are interfaced with the physical environment using input devices (e.g., sensors, analog-to-digital converters) and output devices (e.g., digital-to-analog converters, actuators). Based on the measured values of the physical parameters of interest, the software commands the actuators to apply stimuli to the environment with the purpose of maintaining certain properties in the environment.

Due to their safety-critical nature, getting these systems right is extremely important. A challenging design task is to find the right combination of input devices, output devices, and software such that their integration produces a system that satisfies the requirements. Systems engineers are responsible for this task and, in particular, for choosing the input and output devices. Software engineers must then determine the software part of the system so that the system requirements

* Corresponding author.
E-mail addresses: patcaslm@mcmaster.ca (L.M. Patcas), lawford@mcmaster.ca (M. Lawford), maibaum@mcmaster.ca (T. Maibaum).

http://dx.doi.org/10.1016/j.scico.2015.05.007
0167-6423/

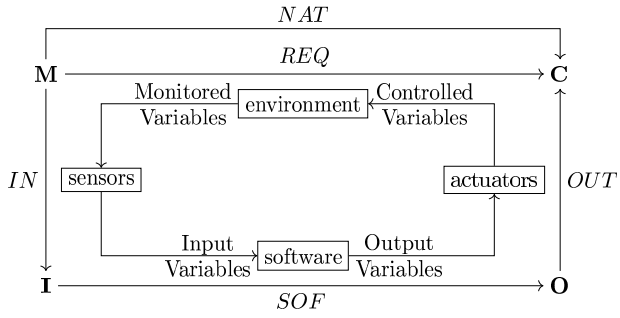


Fig. 1. The four-variable model.

are satisfied. Considering that changes in the specifications of the system requirements and hardware interfaces often arise during the system's development life cycle, the process mentioned above becomes repetitive and thus even more demanding [1], [2, Section 2.6.3]. What if no software can satisfy the constraints imposed by the system requirements and chosen hardware interfaces? Time and resources will be spent trying to develop and verify repeatedly a system that can never satisfy the requirements.

Hence, we ask the following question: is the software part of the system possible at all given a particular choice of hardware interfacing between the system and the physical environment? A positive answer to this question would allow software engineers to proceed with a software design having the confidence that their efforts are not destined to fail from the start. In this case, the requirements of the system are said to be *implementable* with respect to the physical environment and chosen input/output devices, while the software is called *acceptable*. In the case of a negative answer, the next step would be for the systems engineers to understand why that is the case and determine the necessary changes to the specifications of the input and output devices, and possibly to the specification of the system requirements, in order for the software part of the system to become possible. Such a bidirectional interaction between systems engineering and software engineering is stressed in [2, Section 1.2] as being essential in producing dependable software-controlled systems.

In this paper we prove a necessary and sufficient implementability condition for requirements in the four-variable model proposed by Parnas and Madey [3]. This model, depicted in Fig. 1 and described in Section 2, has been used successfully in the development of safety-critical systems in industry and helps to clarify the behaviors of, and the boundaries between, the environment, sensors, actuators, and software. To be implementable, the system requirements must be feasible with respect to the environment (i.e., should specify only behaviors that obey the environmental constraints) and acceptable software behaviors must be possible given the chosen input/output devices. In Section 2, we discuss why the feasibility and acceptability conditions given in Parnas and Madey [3], which may be seen as angelic, are too weak and allow undesirable system and software behaviors. In Section 3 we introduce the demonic calculus of relations [4–6], which will be used to strengthen these conditions in Section 4. Using the strengthened feasibility and acceptability conditions, we will then give a necessary and sufficient implementability condition for the system requirements, along with a mathematical characterization of the software requirements. In Section 5 we describe a detailed analysis of the implementability of the requirements for a pressure sensor trip computer, a subsystem in the shutdown system of a nuclear power plant. This analysis also demonstrates the usefulness of the implementability conditions as rigorous and systematic guiding tools in determining the tolerances needed on the requirements of the pressure sensor trip computer.

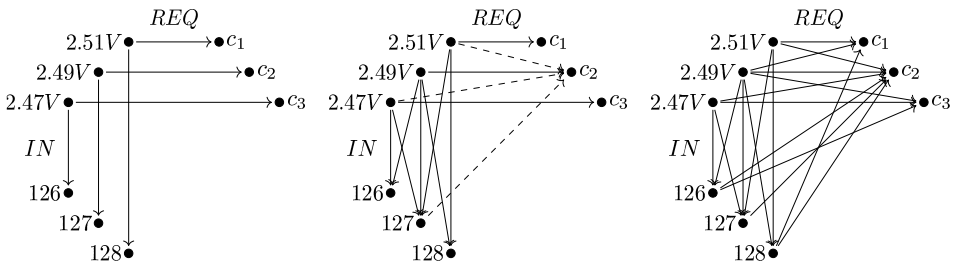
This paper is an extended version of a previous paper by us [7]. Sections 2, 3 and 4 give more details, examples, and proofs; in particular, a more thorough comparison between demonic and angelic semantics is given. Section 5 is completely new.

2. The four-variable model

The model was used as early as 1978 as part of the Software Cost Reduction (SCR) program of the Naval Research Laboratory for specifying the flight software of the U.S. Navy's A-7 aircraft [8]. The ideas from SCR were later extended into the Consortium of Requirements Engineering (CoRE) methodology, which was used for specifying the avionics system of the C-130J military aircraft in the 1980s [9]. Another significant example of a successful use of the four-variable model is the redesign of the software in the shutdown systems of the Darlington nuclear power plant in Ontario, Canada in the 1990s [10–12]. In 2009, the four-variable model was used extensively in the *Requirements Engineering Handbook* [13] that was put together at the request of the U.S. Federal Aviation Administration.

2.1. System requirements and environmental constraints

In the four-variable model, *REQ* models the *system requirements*. At the system requirements level, a system is seen as a black-box that relates physical quantities measured by the system, called *monitored variables*, to physical quantities controlled by the system, called *controlled variables*. For example, monitored variables might be the pressure and temperature inside a nuclear reactor while controlled variables might be visual and audible alarms, as well as the trip signal that initiates



(a) Idealized *REQ* and *IN* (b) *IN* with ±0.02V accuracy (c) *REQ* with tolerances

Fig. 2. Motivational example for a relational four-variable model.

a reactor shutdown; whenever the temperature or pressure reach abnormal values, the alarms go off and the shutdown procedure is initiated. The sets of the possible values for the monitored and controlled variables are denoted by **M** and **C**, respectively. The environmental constraints on the system are described by *NAT* (from "nature"), which restricts the possible values of the monitored and controlled variables. For instance, an environmental constraint might be the maximum rate of climb of an aircraft in the case of an avionics system.

2.2. System design

The possible *system designs* are modeled 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 in the environment to values of *input variables* in the software. The input variables model the information about the environment that is available to the software. For example, *IN* might model a sensor that converts pressure values to analog voltages, which are converted via an analog-to-digital converter (ADC) to integer values stored in a register accessible to the software via an input variable. The *output hardware interface* (digital-to-analog converters and actuators) is modeled by *OUT*, which relates values of the *output variables* of the software to values of controlled variables. An output variable might be, for instance, a boolean variable set by the software with the understanding that the value `true` indicates that a reactor shutdown should occur and the value `false` indicates the opposite. The sets of the possible values of the input and output variables are denoted by **I** and **O**, respectively. Relating values of input variables to values of output variables is *SOF*, which models the *control software* including the input/output device drivers.

The four-variable model is in general relational, not functional. For example, let us consider an input interface *IN* that models an 8-bit resolution ADC which converts monitored voltages m in the range 0–5 V into software input values i according to the formula $i = \lfloor m * 2^8/5 \rfloor$. Fig. 2a depicts *IN* and *REQ* for the monitored voltages $m = 2.47$ V, $m = 2.49$ V, and $m = 2.51$ V. Here, *IN* and *REQ* are functions and model idealized behaviors. If the ADC has a ±0.02 V accuracy (Fig. 2b), then *IN* becomes a relation because, for example, *IN* can produce any of the software input values $i = 126$, $i = 127$, and $i = 128$ for the monitored voltage $m = 2.49$ V. Conversely, the software input $i = 127$ can be the digital representation of any of the monitored voltages $m = 2.47$ V, $m = 2.49$ V, and $m = 2.51$ V. In this example, no system implementation can satisfy the requirements because no matter which system output c_1, c_2, or c_3 is produced by *SOF* together with *OUT* for $i = 127$, this output will violate the requirements (e.g., if c_2 is produced, then $m = 2.47$ V and $m = 2.51$ V will be connected with c_2, something not allowed by *REQ*). A typical engineering approach in such situations is to allow tolerances on the requirements, in which case *REQ* becomes a relation (Fig. 2c) and many system implementations become possible. If hardware inaccuracies are considered for the output interface, then *OUT* will be a relation as well. If we want to capture all the possible behaviors of the control software, then *SOF* will typically have to be a relation. An implementation of *SOF* that runs on an actual computer will be a function (i.e., a deterministic program).

The environmental constraints on the system, *NAT*, are usually described by a relation too. As extreme examples, if everything is possible in the physical environment, then *NAT* is the universal relation between **M** and **C**; if nothing is possible, then *NAT* is the empty relation.

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*.

2.3. Requirements feasibility

From an engineering perspective, the behaviors that do not obey the physical laws of the environment are not implementable. For example, there is no point for the requirements of an autopilot system in an airplane to specify rates of climb higher than what the plane is capable of; such requirements can never be satisfied by a real system and may even be dangerous as they can overstress the engines and airframe. Parnas and Madey [3] used the following two conditions to ensure that the requirements specify behaviors allowed by the environment:

$$\mathrm{dom}\,(NAT) \subseteq \mathrm{dom}\,(REQ)\ ; \tag{1}$$

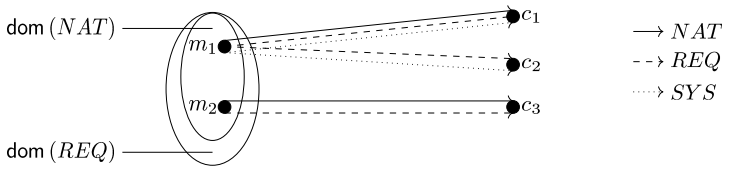


Fig. 3. The acceptability conditions of Parnas and Madey [3] are too weak.

$$\mathsf{dom}\,(REQ \cap NAT) = \mathsf{dom}\,(REQ) \cap \mathsf{dom}\,(NAT)\ . \tag{2}$$

Condition (1) asks *REQ* to specify system response for all the monitored environmental states allowed by *NAT*. Under normal circumstances, it can be assumed that the environment will never be in a state that is not contained in the domain of *NAT*. Condition (2) constrains *REQ* to agree with *NAT* on at least one output for every input *REQ* and *NAT* have in common. Together, conditions (1) and (2) ensure that, for every monitored state allowed by the environment, the requirements ask the system to produce at least one output that is allowed by *NAT*. If the two conditions are satisfied, *REQ* is said to be *feasible* with respect to *NAT* [3].

2.4. Software acceptability

Assuming that the requirements are feasible with respect to the environment, Parnas and Madey [3] proposed the following *acceptability* condition for the software:

$$NAT \cap (IN \fatsemi SOF \fatsemi OUT) \subseteq REQ\ . \tag{3}$$

Here, the operator $\fatsemi$ is the usual composition of relations.

A system implementation $SYS = IN \fatsemi SOF \fatsemi OUT$ is then acceptable if and only if it satisfies the following condition:

$$NAT \cap SYS \subseteq REQ\ . \tag{4}$$

These acceptability conditions are, however, not strong enough. Let us consider the relations *NAT*, *REQ*, and *SYS* in Fig. 3, where m_1, m_2 are possible monitored environmental states and c_1, c_2, c_3 are possible controlled environmental states. These relations satisfy the conditions (1), (2) and (4). Therefore the system requirements *REQ* are feasible with respect to *NAT*, and the system implementation *SYS* and software *SOF* are deemed acceptable although they should not be. This is the case due to the following problems.

2.4.1. Problem 1

In Fig. 3, $(m_1, c_2) \in SYS$, $(m_1, c_2) \in REQ$, and $(m_1, c_2) \notin NAT$. That is, a system implementation *SYS* that drives the environment into states that violate environmental laws specified by *NAT* is deemed acceptable. From an engineering perspective, such implementations are not possible and it is important to reject early specifications that allow them. A similar problem with the acceptability conditions proposed by Parnas and Madey was pointed out by Gunter et al. [14].

The cause of this problem is the fact that the requirements allow the pair (m_1, c_2) while the environment does not allow it. For more complex requirements, such cases may not be obvious and the system designers may attempt to implement behaviors that are not allowed by the environment. For the feasibility and acceptability conditions proposed by Parnas and Madey [3] to work as intended, only right choices will have to be made during design. In this sense, these conditions may be seen as angelic.

In Section 4.1 we will offer a solution to this problem using a demonic approach that does not allow requirements specifications such as the one in Fig. 3. We will strengthen the condition for the feasibility of requirements such that throughout the whole domain of *NAT*, the requirements are allowed to specify only behaviors allowed by *NAT*.

2.4.2. Problem 2

In Fig. 3, $(m_2, c_3) \in NAT$, $(m_2, c_3) \in REQ$, and $(m_2, c_3) \notin SYS$. That is, a system implementation *SYS* that does not react to all possible environmental states denoted by $\mathsf{dom}\,(NAT)$ is deemed acceptable.

This is another manifestation of the angelic nature of the acceptability conditions proposed by Parnas and Madey [3]. These conditions work as intended only if the designers do not make bad choices. However, designers will make bad choices if allowed to do so, even if unintentionally. In Section 4.2 we will give a stronger, demonic acceptability condition that rejects specifications of *SYS* such as the one in Fig. 3.

2.4.3. Problem 3

Another problem with the acceptability conditions (3) and (4) proposed by Parnas and Madey [3] is that they allow implementations that may sometimes not return any result for an input as long as there is the chance that some other times they produce expected results for that input. This can happen when $\mathsf{ran}\,(IN) \supset \mathsf{dom}\,(SOF)$ or $\mathsf{ran}\,(SOF) \supset \mathsf{dom}\,(OUT)$ because *SOF* will not react to all values produced by *IN* and *OUT* will not react to all values produced by *SOF*. Deeming

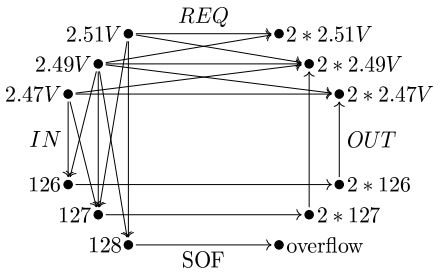


Fig. 4. Angelic semantics allows undesirable implementations.

such implementations acceptable is indicative of an angelic view since the implicit assumption is that somehow only good choices will be made at runtime and the implementation will always return expected results.

Let us consider an example that illustrates this point. Let a relation *IN* model an 8-bit resolution ADC which converts monitored voltages m in the range 0–5 V into software input values i according to the formula $i = \lfloor m * 2^8/5 \rfloor$. The requirements ask the system to produce at the output the double of the input with a tolerance of ± 0.04 V. Because the relation *NAT* says that the monitored voltages will be in the range 0–2.49 V, it is decided that 8-bit unsigned integers will be used to represent the values of the output variable $o = 2 * i$ set by the software. If the converter has an accuracy of ± 0.02 V, the following situation depicted in Fig. 4 can occur: for $m = 2.49$ V, which is a voltage allowed by *NAT*, the input hardware *IN* may produce any of the software inputs $i = 126$, $i = 127$ and $i = 128$; for $i = 126$ and $i = 127$ the system returns outputs allowed by *REQ*, but for $i = 128$ the corresponding software output does not fit in the 8-bit unsigned integer variable and an overflow occurs resulting in a runtime error or in an incorrect value. In either case, none of the results expected by *REQ* is produced. The acceptability condition (3) of Parnas and Madey is trivially satisfied although an implementation $IN ; SOF ; OUT$ could produce an overflow: $NAT \cap (IN ; SOF ; OUT) = \{(2.49\text{ V}, 2 * 2.49\text{ V})\} \subseteq REQ = \{(2.49\text{ V}, 2 * 2.49\text{ V}), (2.51\text{ V}, 2 * 2.51\text{ V}), (2.53\text{ V}, 2 * 2.53\text{ V})\}$.

The root of this problem is that the software does not have any control over what values it receives from the input devices. Similarly, the output devices do not have control over the output values produced by the software. As such, it would be more sensible to assume that bad things will happen if given the opportunity, and reject specifications that allow such "demonic" opportunities, instead of relying on an angel that always makes good choices. We will address this issue in Section 4.2 by redefining the acceptability condition of Parnas and Madey[3] using the demonic calculus of relations.

3. The demonic calculus of relations

In this section we introduce the demonic calculus of relations [4–6]. This calculus will be used in Section 4 to rectify the problems with requirements feasibility and software acceptability described in Section 2.

In addition to the abstract algebraic notation customary in the relation algebra literature and somewhat standardized in [15], we will also give whenever convenient the equivalent, but more verbose, set-theoretic notation. We favor the set-theoretic notation since we believe that it is more accessible to the intended (software) engineering audience.

Relation algebras are particular cases of Boolean algebras and are frequently introduced using the language of universal algebra [15, Section 1.4] or that of category theory [6]. Again, to be more accessible to engineers, we will use a particular case of a relation algebra, the algebra of concrete binary relations.

A concrete binary *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 ordered pairs (a, b), where $a \in A$ and $b \in B$. The sets A and B are called the *source* and, respectively, the *target* of relation R. For describing concrete relations we will use the usual set comprehension, or set builder, notation. In this notation, a relation $R \subseteq A \times B$ is given as $R = \{(a, b) \in A \times B | R_{pred}(a, b)\}$, where R_{pred}, called the *characteristic predicate* of relation R, is a predicate that describes the constraints that a pair (a, b) has to satisfy to be part of R.

Some elementary operations involving a relation $R \subseteq A \times B$ are:

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

A relation $R \subseteq A \times B$ is *univalent* if every element in its domain is mapped 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*.

As seen in Fig. 2, the inaccuracy of the input and output hardware interfaces introduces uncertainty in a system implementation. Likewise, tolerances on system requirements give potential implementations a number of equally acceptable choices for producing a result. Uncertainty and choice are forms of nondeterminism. Non-univalent relations are natural

candidates for modeling nondeterminism: the image set of an element in the domain of a non-univalent relation denotes all the acceptable results for that input. Functional relations model deterministic behaviors since the image sets of the elements in their domains are singletons. In addition to the nondeterminism caused by input/output hardware inaccuracies and tolerances on requirements, in the four-variable model there is another form of nondeterminism caused by the composition of partial specifications for *IN*, *SOF*, and *OUT*, as seen in Section 2.4.3.

Various approaches to nondeterminism have been studied in variations of Dijkstra's weakest-precondition calculus [16–20], as well as in relation-algebraic approaches to formal specification and program semantics [21,22,4,23,20,5,6]. The main approaches to deal with nondeterministic specifications are angelic and demonic. In the angelic approach, specifications that allow "bad" behaviors for some inputs are permitted as long as they also allow "good" behaviors for those inputs. In contrast, in the demonic approach, specifications that allow "bad" behaviors are not permitted at all. Since the four-variable model has been used traditionally in the safety-critical domain, we argue that a demonic approach is more suitable and use the demonic calculus of relations [4–6]. Because the operators in the demonic calculus are defined in terms of their angelic counterparts, we first present the angelic operators and then the demonic operators.

3.1. Angelic operators

The angelic operators are the usual relational operators.

The *intersection* of two relations $P \subseteq A \times B$ and $Q \subseteq A \times B$ is the relation $P \cap Q = \{(a, b) \in A \times B \mid (a, b) \in P \wedge (a, b) \in Q\}$. Their *union* is $P \cup Q = \{(a, b) \in A \times B \mid (a, b) \in P \vee (a, b) \in Q\}$.

The relation P is *contained* in the relation Q, written $P \subseteq Q$, if and only if $\forall a \in A \bullet \forall b \in B \bullet (a, b) \in P \Rightarrow (a, b) \in Q$. Relational containment, or *inclusion*, is a partial order that induces a complete lattice structure on the set of relations between A and B. The join operation on this lattice is $\cup$ and the meet operation is $\cap$. The top element is the relation $\top_{A,B} = \{(a, b) \in A \times B \mid \mathsf{true}\}$, called the *universal relation* between A and B, and the bottom element is the *empty relation* $\bot_{A,B} = \{(a, b) \in A \times B \mid \mathsf{false}\}$.

Relational inclusion is used as a *refinement* ordering in, for example, [22,24,25] and is known as *partial correctness* in Kahl [6], where an elegant mathematical explanation is given as to why the satisfaction and refinement concepts are the same when relations are used for describing both specifications and implementations. Therefore, in this paper we will use "satisfies", "refines", and "implements" interchangeably when describing the relationship between implementations and specifications.

The meaning of the statement "P implements R" in the relational inclusion sense is as follows:

- if R is not defined for some inputs (i.e., R is a partial relation), then those inputs are considered illegal and P must not produce any results for them;
- for the inputs for which R is defined, P may or may not produce a result, but if P produces a result, then that result must be allowed by R (i.e., relational inclusion is angelic). A degenerate case is the empty relation, which satisfies any specification (the empty relation is the bottom element in the lattice induced by $\subseteq$).

Allowing implementations that are not required to deal with all the inputs in the domain of their specifications is problematic for safety-critical systems. Moreover, allowing the empty relation to be an acceptable implementation of any specification means that implementations that do not produce any results are always acceptable. This is also not something desirable, especially for a safety-critical system.

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

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

Relational composition is an associative operation.

The precedence of the relational operators introduced so far is as follows: the unary operators $^\smile$ and $\overline{\ }$ are evaluated first; the binary operator $;$ is evaluated next; the binary operators $\cap$ and $\cup$ are evaluated last.

The relational composition and inclusion operations induce two residuation operations, the left and right residuals [22, 24–26,4,15,6]. The residuals are useful when a specification is refined by a composition of two other specifications of which one is not known, and will be used in Sections 3.3 and 4 in proving an existence condition for an acceptable relation *SOF* in the four-variable model.

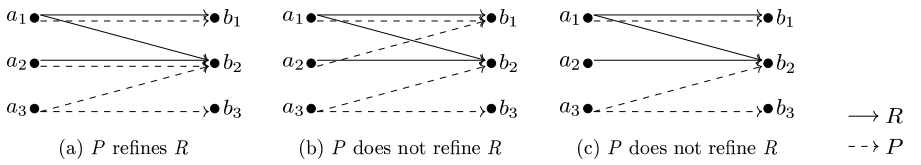
Assuming two relations $R \subseteq A \times C$ and $Q \subseteq B \times C$, the *left residual* of R by Q, denoted R/Q, is the largest solution of the inequality $X ; Q \subseteq R$, where $X \subseteq A \times B$ is the unknown:

$$X ; Q \subseteq R \Leftrightarrow X \subseteq R/Q\ .$$

The value of the left residual of R by Q is:

$$R/Q = \overline{\overline{R} ; Q^\smile} = \{(a, b) \in A \times B \mid \forall c \in C \bullet (b, c) \in Q \Rightarrow (a, c) \in R\} = \{(a, b) \in A \times B \mid Q(b) \subseteq R(a)\}.$$

Given two relations $R \subseteq A \times C$ and $P \subseteq A \times B$, the *right residual* of R by P, denoted $P \backslash R$, is the largest solution of the inequality $P ; X \subseteq R$, where $X \subseteq B \times C$ is the unknown:



(a) P refines R (b) P does not refine R (c) P does not refine R

Fig. 5. Examples of demonic refinement.

$$P ; X \subseteq R \Leftrightarrow X \subseteq P \backslash R .$$

The value of the right residual of R by P is:

$$P \backslash R = \overline{P^{\smile} ; \overline{R}} = \{(b, c) \in B \times C \mid \forall a \in A \bullet (a, b) \in P \Rightarrow (a, c) \in R\} = \{(b, c) \in B \times C \mid P^{\smile}(b) \subseteq R^{\smile}(c)\} .$$

The precedence of / and \ is the same as the precedence of the relational composition. The residuation operations are loosely analogous to division of natural numbers and the values of the residuals are a form of quotient. The left residual R/Q can be understood as what remains on the left of R after R is "divided" by Q on the right. Dually, the right residual $P\backslash R$ is what remains on the right of R after "dividing" R by P on the left. Hoare and He [22,24] were among the first to advocate the importance of the relational residuals to software development.[1] Hoare and He called the left residual R/Q the *weakest prespecification* of program Q to achieve specification R, and the right residual $P\backslash R$ the *weakest postspecification* of program P to achieve specification R.

3.2. Demonic operators

We now present the demonic relational operators that will be used in the paper and motivate their suitability for safety-critical systems compared to their angelic counterparts.

First, we introduce constructs that will allow us to work with partial relations and also to make the transition from the abstract relation-algebraic presentation typical in the literature to a set-theoretic presentation. The rationale for allowing partial relations is that in practice, in the early stages of system development formulating complete specifications for complex systems is virtually an impossible task; the specifications are iteratively refined, adding more detail as the system becomes better understood until the specifications cover all the possible cases that can arise [27–29]. Before getting to that point, however, many useful analyses can be performed, such as checking if an acceptable implementation really is possible. From the perspective of validation and verification, working with partial relations is a pragmatic approach: if we cannot satisfy a partial specification, we will not be able to meet a more complete version of that specification. The *domain restriction* of a relation $P \subseteq A \times B$ to a set $A' \subseteq A$ is the relation $P\big|_{A'} = \{(a, b) \in P \mid a \in A'\}$. The *range restriction* of a relation $P \subseteq A \times B$ to a set $B' \subseteq B$ is the relation $P\big|^{B'} = \{(a, b) \in P \mid b \in B'\}$. The domain and range restrictions are also known as the prerestriction and, respectively, postrestriction constructs in [30]. We will use the following domain and range restrictions:

- the domain restriction of $P \subseteq A \times B$ to the domain of $R \subseteq A \times C$:

$$P\big|_{\mathsf{dom}(R)} = P \cap R ; \top_{C,B} = \{(a, b) \in P \mid a \in \mathsf{dom}(R)\};$$

- the domain restriction of $P \subseteq A \times B$ to the range of $R \subseteq C \times A$:

$$P\big|_{\mathsf{ran}(R)} = P \cap R^{\smile} ; \top_{C,B} = \{(a, b) \in P \mid a \in \mathsf{ran}(R)\};$$

- the range restriction of $P \subseteq A \times B$ to the domain of $R \subseteq B \times C$:

$$P\big|^{\mathsf{dom}(R)} = P \cap \top_{A,C} ; R^{\smile} = \{(a, b) \in P \mid b \in \mathsf{dom}(R)\}.$$

3.2.1. Demonic refinement

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 $P\big|_{\mathsf{dom}(R)} \subseteq R$. Consider the relations P and R in Fig. 5: in Fig. 5a, P refines R; in Fig. 5b, P does not refine R because $(a_2, b_1) \in P$ but $(a_2, b_1) \notin R$; and in Fig. 5b, P does not refine R because $\mathsf{dom}(R) \not\subseteq \mathsf{dom}(P)$.

Maddux [20] made the connection between the approaches to nondeterminism and stepwise refinement in variations of Dijkstra's weakest precondition calculus [16,18,17,19] and those in relation-algebraic formalisms [31,30,32,33,4,34,5,6].[2] This connection reveals that demonic refinement appears under different guises in the literature: "more defined than" [30,32]; total correctness [31,5,6]; demonic refinement [33,4,34,5,6].

[1] Hoare and He use a different notation than ours; we follow the RelMiCS [15] conventions.

[2] The reader should note that the demonic refinement ordering in [32,4] is the converse of the usual demonic ordering.

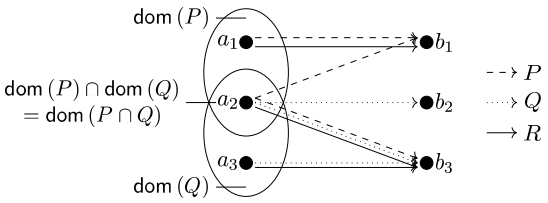


Fig. 6. Example of demonic intersection.

Demonic refinement is a partial order and induces a complete join semi-lattice, usually referred to as the *demonic lattice* [32,33,5,6]. The top element of the demonic lattice is the empty relation $\perp\!\!\!\perp$, which does not impose any constraints whatsoever on its implementations. As such, any relation is a refinement of $\perp\!\!\!\perp$. The sub-lattice between $\perp\!\!\!\perp$ and the universal relation $\top\!\!\!\top$ is the set of partial relations, which specify termination only for the inputs in their domain. Below $\top\!\!\!\top$, inclusively, is the set of the total relations which specify termination everywhere; the minima of this set are the ideal implementations, which are mappings (i.e., total functions). The join operation of the demonic lattice is the demonic union $\sqcup$ and will not be used in this paper. The meet operation is the demonic intersection $\sqcap$, which is not always well defined (the demonic lattice is a join semi-lattice).

The meaning of the statement "P implements R" in the demonic refinement sense is as follows:

- for every input for which R is defined, P must produce only outputs allowed by R (i.e., an implementation is at least as deterministic as its specification);
- for the inputs for which R is not defined, P is allowed to do anything (i.e., P may produce incorrect results or no result at all).

Compared to angelic refinement, demonic refinement does not allow empty implementations if the specification is not empty. Moreover, demonic refinement forces an implementation to deal with all the inputs in the domain of its specification. These differences make the demonic refinement better suited for a safety-critical setting. It is debatable, however, in the case of demonic refinement, if allowing arbitrary behavior outside the domain of a specification is the best thing to do. When we give a demonic semantics to the four-variable model in Section 4, we will explain how this can be dealt with in practice. As particular cases, if P and R are total or if $\mathsf{dom}\,(P) = \mathsf{dom}\,(R)$, then $P \sqsubseteq R$ and $P \subseteq R$ are equivalent.

3.2.2. Demonic intersection

Two relations $P \subseteq A \times B$ and $Q \subseteq A \times B$ are *compatible* if and only if

$$\mathsf{dom}\,(P) \cap \mathsf{dom}\,(Q) \subseteq \mathsf{dom}\,(P \cap Q)\,. \tag{5}$$

Condition (5) means that for every input in common, P and Q should have at least one output in common.

The *demonic intersection* of P and Q, denoted as $P \sqcap Q$, is defined only if P and Q are compatible. If the demonic intersection of P and Q is defined, then its value is:

$$P \sqcap Q \succeq (P \cap Q) \cup (P \cap \overline{Q \fatsemi \top\!\!\!\top}) \cup (\overline{P \fatsemi \top\!\!\!\top} \cap Q) = (P \cap Q) \cup P\big|_{\overline{\mathsf{dom}(Q)}} \cup Q\big|_{\overline{\mathsf{dom}(R)}}\,. \tag{6}$$

The symbol $\succeq$, called the "venturi tube" [6], has the following meaning: for any two expressions ϕ and ψ, $\phi \succeq \psi$ means that if ϕ is defined, then ψ is defined and equal to ϕ. The intuition for (6) is that $P \sqcap Q$ captures the behavior that is common to both P and Q; outside the domain of Q, $P \sqcap Q$ does exactly what P does; and, outside the domain of P, $P \sqcap Q$ does exactly what Q does. For example, let $P = \{(a_1, b_1), (a_2, b_1), (a_2, b_3)\}$ and $Q = \{(a_2, b_2), (a_2, b_3), (a_3, b_3)\}$. In this case, $P \sqcap Q = \{(a_2, b_3), (a_1, b_1), (a_3, b_3)\}$ (Fig. 6).

3.2.3. Demonic composition

The *demonic composition* of two relations $P \subseteq A \times B$ and $Q \subseteq B \times C$ is defined as:

$$P \boxdot Q = P \fatsemi Q \cap \overline{P \fatsemi \overline{Q \fatsemi \top\!\!\!\top_{C,C}}} = \{(a, c) \in P \fatsemi Q \mid P(a) \subseteq \mathsf{dom}\,(Q)\}\,.$$

Demonic composition is the same as the angelic composition when P is univalent or when Q is total. The difference between these two notions of relational composition is indicative of the difference between angelic and demonic semantics. As an example, let us consider the following two relations $P = \{(a_1, b_1), (a_1, b_2)\}$ and $Q = \{(b_1, c_1)\}$, depicted in Fig. 7. Here, $P \fatsemi Q$ allows the dead end (a_1, b_2) as long as there is a chance that c_1 will be reached via b_1. On the other hand, $P \boxdot Q$ is empty because there is the possibility that an implementation will get stuck at b_2 and will not reach c_1.

3.2.4. Demonic residuals

As was the case with angelic composition and angelic inclusion, demonic composition and demonic refinement induce two residuation operations, the demonic left and right residuals. The demonic residuals are useful when a relation is refined by a demonic composition of two other relations and one of these relations is not known. They will be used in

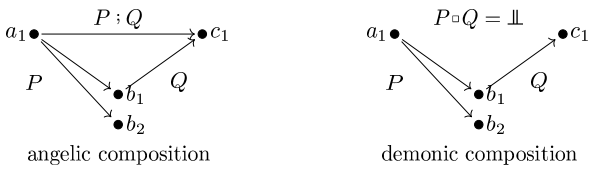


Fig. 7. Demonic vs. angelic composition.

Sections 3.3 and 4 to prove a necessary and sufficient condition for the existence of a relation *SOF* such that the diagram of the four-variable model commutes.

The *demonic left residual*[3] of a relation $R \subseteq A \times C$ by a relation $Q \subseteq B \times C$, denoted $R \mathbin{/\!\!/} Q$, is defined as the largest solution with respect to $\sqsubseteq$ of the inequation $X \sqcap Q \sqsubseteq R$, where $X \subseteq A \times B$ is the unknown:

$$X \sqcap Q \sqsubseteq R \Leftrightarrow X \sqsubseteq R \mathbin{/\!\!/} Q \,.$$

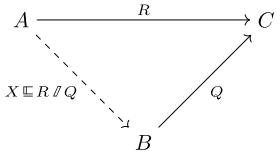
A solution X, called a *demonic left factor* of R through Q, does not always exist. As such, the demonic left residual $R \mathbin{/\!\!/} Q$ is not always defined. For example, if $Q = \perp\!\!\!\perp_{B,C}$ and $R \neq \perp\!\!\!\perp_{A,C}$, then $X \sqcap Q = \perp\!\!\!\perp_{A,C}$ for any $X \subseteq A \times B$, in which case $X \sqcap Q$ is not a demonic refinement of R and $R \mathbin{/\!\!/} Q$ is not well defined.

If $R \mathbin{/\!\!/} Q$ is defined, then its value is:

$$R \mathbin{/\!\!/} Q \succeq R/Q \cap \top_{A,C} ; Q^{\smile} = (R/Q)\,\big|^{\mathsf{dom}(Q)} = \{(a,b) \in A \times B \mid b \in \mathsf{dom}(Q) \wedge Q(b) \subseteq R(a)\}\,. \tag{7}$$

We now state and prove a necessary and sufficient condition for the existence of a demonic left factor and, therefore, for the definedness of the demonic left residual.

Lemma 1. *Given two relations $R \subseteq A \times C$ and $Q \subseteq B \times C$, there exists a demonic left factor $X \subseteq A \times B$ such that $X \sqcap Q \sqsubseteq R$ if and only if $\forall a \in \mathsf{dom}(R) \mathbin{.} \exists b \in \mathsf{dom}(Q) \mathbin{.} Q(b) \subseteq R(a)$.*



Proof. If direction:

$$\begin{aligned}
&\exists X \mathbin{.} X \sqcap Q \sqsubseteq R \\
\Rightarrow\ &\langle \text{by definition, if } X \sqcap Q \sqsubseteq R \text{ admits a solution in } X, \text{ then } R \mathbin{/\!\!/} Q \text{ also is a solution} \rangle \\
&(R \mathbin{/\!\!/} Q) \sqcap Q \sqsubseteq R \\
\Rightarrow\ &\langle \text{by definition of } \sqsubseteq \rangle \\
&\mathsf{dom}(R) \subseteq \mathsf{dom}((R \mathbin{/\!\!/} Q) \sqcap Q) \\
\Rightarrow\ &\mathsf{dom}(R) \subseteq \mathsf{dom}(R \mathbin{/\!\!/} Q) \\
\Rightarrow\ &\forall a \in \mathsf{dom}(R) \mathbin{.} \exists b \in B \mathbin{.} (a,b) \in R \mathbin{/\!\!/} Q \\
\Rightarrow\ &\langle \text{by (7)} \rangle \\
&\forall a \in \mathsf{dom}(R) \mathbin{.} \exists b \in \mathsf{dom}(Q) \mathbin{.} Q(b) \subseteq R(a)\,.
\end{aligned}$$

Only if direction:

$$\begin{aligned}
&\forall a \in \mathsf{dom}(R) \mathbin{.} \exists b \in \mathsf{dom}(Q) \mathbin{.} Q(b) \subseteq R(a) \\
\Leftrightarrow\ &\langle \text{by (7)} \\
&\forall a \in \mathsf{dom}(R) \mathbin{.} \exists b \in \mathsf{dom}(Q) \mathbin{.} (a,b) \in R \mathbin{/\!\!/} Q \\
\Leftrightarrow\ &\forall a \in \mathsf{dom}(R) \mathbin{.} \exists b \in B \mathbin{.} (a,b) \in R \mathbin{/\!\!/} Q \wedge b \in \mathsf{dom}(Q)
\end{aligned}$$

[3] The demonic left residual is called the conjugate kernel in [35].

$$
\begin{aligned}
&\Leftrightarrow \langle \text{by definition of } \square \rangle \\
&\quad \forall a \in \mathsf{dom}\,(R) \mathbin{.} \exists c \in C \mathbin{.} (a, c) \in (R \mathbin{/\!\!/} Q) \mathbin{\square} Q \\
&\Rightarrow \mathsf{dom}\,(R) \subseteq \mathsf{dom}\,((R \mathbin{/\!\!/} Q) \mathbin{\square} Q)\ . \qquad (8) \\
&\quad ((R \mathbin{/\!\!/} Q) \mathbin{\square} Q)\,\big|_{\mathsf{dom}(R)} \subseteq R \\
&\Leftrightarrow \forall a \in \mathsf{dom}\,(R) \mathbin{.} ((R \mathbin{/\!\!/} Q) \mathbin{\square} Q)\,(a) \subseteq R(a) \\
&\Leftrightarrow \langle \text{by unfolding } \subseteq \rangle \\
&\quad \forall a \in \mathsf{dom}\,(R) \mathbin{.} \forall c \in C \mathbin{.} (a, c) \in ((R \mathbin{/\!\!/} Q) \mathbin{\square} Q) \Rightarrow (a, c) \in R \\
&\Leftrightarrow \langle \text{by definition of } \square \text{ and } ; \rangle \\
&\quad \forall a \in \mathsf{dom}\,(R) \mathbin{.} \forall c \in C \mathbin{.} (\exists b \in \mathsf{dom}\,(Q) \mathbin{.} (a, b) \in R \mathbin{/\!\!/} Q \wedge (b, c) \in Q) \Rightarrow (a, c) \in R \\
&\Leftrightarrow \langle \text{by (7)} \rangle \\
&\quad \forall a \in \mathsf{dom}\,(R) \mathbin{.} \forall c \in C \mathbin{.} (a, c) \in R \Rightarrow (a, c) \in R \\
&\Leftrightarrow \mathsf{true}\,. \qquad (9) \\
&\quad \forall a \in \mathsf{dom}\,(R) \mathbin{.} \exists b \in \mathsf{dom}\,(Q) \mathbin{.} Q\,(b) \subseteq R(a) \\
&\Rightarrow \langle \text{by (8) \& (9) \& definition of } \sqsubseteq \rangle \\
&\quad (R \mathbin{/\!\!/} Q) \mathbin{\square} Q \sqsubseteq R \\
&\Rightarrow \langle \text{by taking } X = R \mathbin{/\!\!/} Q \rangle \\
&\quad \exists X \mathbin{.} X \mathbin{\square} Q \sqsubseteq R\,. \qquad \square
\end{aligned}
$$

Several conditions for the definedness of the demonic left residual can be found in the literature, such as, if converted to our notation: $\mathsf{dom}\,(R) \subseteq \mathsf{dom}\left((R/Q)\,\big|^{\mathsf{dom}(Q)}\right)$ in [33,4] and $\mathsf{dom}\,(R) \subseteq \mathsf{dom}\,((R/Q) ; Q)$ in [5,6]. It can be shown that these conditions are equivalent to our condition in Lemma 1.

The *demonic right residual* of a relation $R \subseteq A \times C$ by a relation $P \subseteq A \times B$, denoted $P \mathbin{\backslash\!\!\backslash} R$, is defined as the largest solution with respect to $\sqsubseteq$ of the inequation $P \mathbin{\square} X \sqsubseteq R$, where $X \subseteq B \times C$ is the unknown:

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

A solution X, called a *demonic right factor* of R through P, does not always exists. Therefore, the demonic right residual $P \mathbin{\backslash\!\!\backslash} R$ is not always defined.

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

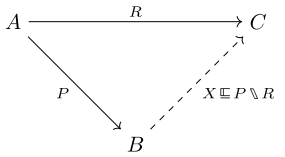
$$
\begin{aligned}
P \mathbin{\backslash\!\!\backslash} R &\succeq \left(P \cap R ; \top_{C,B}\right) \backslash R \cap \left(P \cap R ; \top_{C,B}\right)^{\smile} ; \top_{A,C} \\
&\succeq \left(P\big|_{\mathsf{dom}(R)} \Big\backslash R\right)\Big|_{\mathsf{ran}\left(P|_{\mathsf{dom}(R)}\right)} \\
&\succeq \left\{(b, c) \in B \times C \;\middle|\; b \in \mathsf{ran}\left(P\big|_{\mathsf{dom}(R)}\right) \wedge \left(P\big|_{\mathsf{dom}(R)}\right)^{\smile}(b) \subseteq R^{\smile}(c)\right\}. \qquad (10)
\end{aligned}
$$

We now give a necessary and sufficient condition for the existence of a demonic right factor and, therefore, for the definedness of the demonic right residual.

Lemma 2. *Given two relations $R \subseteq A \times C$ and $P \subseteq A \times B$, there exists a demonic right factor $X \subseteq B \times C$ such that $P \mathbin{\square} X \sqsubseteq R$ if and only if the following conditions are both satisfied:*

(i) $\mathsf{dom}\,(R) \subseteq \mathsf{dom}\,(P)$;

(ii) $\forall b \in \mathsf{ran}\left(P\big|_{\mathsf{dom}(R)}\right) \mathbin{.} \exists c \in C \mathbin{.} \left(P\big|_{\mathsf{dom}(R)}\right)^{\smile}(b) \subseteq R^{\smile}(c)$.



Proof. If direction:

$$
\begin{aligned}
& \exists X \mathbin{.} P \mathbin{\square} X \mathbin{⫦} R \\
\Rightarrow\ & \langle \text{by definition, if } P \mathbin{\square} X \mathbin{⫦} R \text{ has a solution in } X, \text{ then } P \setminus R \text{ also is a solution} \rangle \\
& P \mathbin{\square} (P \setminus R) \mathbin{⫦} R \\
\Rightarrow\ & \langle \text{by definition of } ⫦ \rangle \\
& \mathsf{dom}\,(R) \subseteq \mathsf{dom}\,(P \mathbin{\square} (P \setminus R)) \\
\Rightarrow\ & \langle \text{by definition of } \square \rangle \\
& \mathsf{dom}\,(R) \subseteq \mathsf{dom}\,(P) \wedge \mathsf{ran}\left(P\big|_{\mathsf{dom}(R)}\right) \subseteq \mathsf{dom}\,(P \setminus R) \\
\Rightarrow\ & \mathsf{dom}\,(R) \subseteq \mathsf{dom}\,(P) \wedge \forall b \in \mathsf{ran}\left(P\big|_{\mathsf{dom}(R)}\right) \mathbin{.} \exists c \in C \mathbin{.} (b, c) \in P \setminus R \\
\Rightarrow\ & \langle \text{by (10)} \rangle \\
& \mathsf{dom}\,(R) \subseteq \mathsf{dom}\,(P) \wedge \forall b \in \mathsf{ran}\left(P\big|_{\mathsf{dom}(R)}\right) \mathbin{.} \exists c \in C \mathbin{.} \left(P\big|_{\mathsf{dom}(R)}\right)^{\smile} (b) \subseteq R^{\smile}(c)\,.
\end{aligned}
$$

Only if direction:

$$
\begin{aligned}
& \mathsf{dom}\,(R) \subseteq \mathsf{dom}\,(P) \wedge \forall b \in \mathsf{ran}\left(P\big|_{\mathsf{dom}(R)}\right) \mathbin{.} \exists c \in C \mathbin{.} \left(P\big|_{\mathsf{dom}(R)}\right)^{\smile} (b) \subseteq R^{\smile}(c) \\
\Rightarrow\ & \langle \text{by (10)} \rangle \\
& \mathsf{dom}\,(R) \subseteq \mathsf{dom}\,(P) \wedge \forall b \in \mathsf{ran}\left(P\big|_{\mathsf{dom}(R)}\right) \mathbin{.} \exists c \in C \mathbin{.} (b, c) \in P \setminus R \\
\Rightarrow\ & \forall a \in \mathsf{dom}\,(R) \mathbin{.} \forall b \in P(a) \mathbin{.} \exists c \in C \mathbin{.} (b, c) \in P \setminus R \\
\Rightarrow\ & \langle \text{by definition of } \square \rangle \\
& \forall a \in \mathsf{dom}\,(R) \mathbin{.} \exists c \in C \mathbin{.} (a, c) \in P \mathbin{\square} (P \setminus R) \\
\Rightarrow\ & \mathsf{dom}\,(R) \subseteq \mathsf{dom}\,(P \mathbin{\square} (P \setminus R))\,. \qquad (11)
\end{aligned}
$$

$$
\begin{aligned}
& (P \mathbin{\square} (P \setminus R))\big|_{\mathsf{dom}(R)} \subseteq R \\
\Leftrightarrow\ & \forall a \in \mathsf{dom}\,(R) \mathbin{.} (P \mathbin{\square} (P \setminus R))\,(a) \subseteq R(a) \\
\Leftrightarrow\ & \langle \text{by unfolding } \subseteq \rangle \\
& \forall a \in \mathsf{dom}\,(R) \mathbin{.} \forall c \in C \mathbin{.} (a, c) \in P \mathbin{\square} (P \setminus R) \Rightarrow (a, c) \in R \\
\Leftrightarrow\ & \langle \text{by unfolding } \square \text{ and } ; \rangle \\
& \forall a \in \mathsf{dom}\,(R) \mathbin{.} \forall c \in C \mathbin{.} (\exists b \in \mathsf{dom}\,(P \setminus R) \mathbin{.} (a, b) \in P \wedge (b, c) \in P \setminus R) \Rightarrow (a, c) \in R \\
\Leftrightarrow\ & \langle \text{by (10)} \rangle \\
& \forall a \in \mathsf{dom}\,(R) \mathbin{.} \forall c \in C \mathbin{.} (a, c) \in R \Rightarrow (a, c) \in R \\
\Leftrightarrow\ & \mathsf{true}\,. \qquad (12)
\end{aligned}
$$

$$
\begin{aligned}
& \mathsf{dom}\,(R) \subseteq \mathsf{dom}\,(P) \ \wedge \ \forall b \in \mathsf{ran}\left(P\big|_{\mathsf{dom}(R)}\right) \mathbin{.} \exists c \in C \mathbin{.} \left(P\big|_{\mathsf{dom}(R)}\right)^{\smile} (b) \subseteq R^{\smile}(c) \\
\Rightarrow\ & \langle \text{by (11) \& (12) \& definition of } ⫦ \rangle \\
& P \mathbin{\square} (P \setminus R) \mathbin{⫦} R \\
\Rightarrow\ & \langle \text{by taking } X = P \setminus R \rangle \\
& \exists X \mathbin{.} P \mathbin{\square} X \mathbin{⫦} R\,. \qquad \square
\end{aligned}
$$

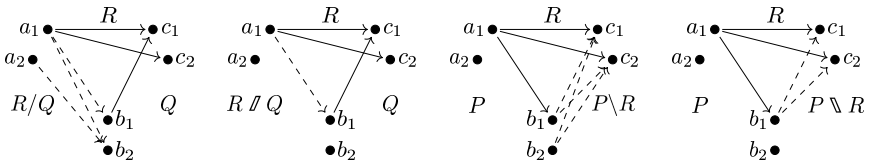


Fig. 8. Demonic vs. angelic residuals.

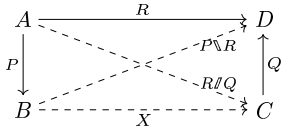


Fig. 9. Existence of a demonic mid factor.

The definedness conditions for the demonic right residual presented in the literature [4,33,6] can be converted to the following common form in our notation: $\mathsf{dom}(R) \subseteq \mathsf{dom}(P) \wedge \mathbb{T}_{B,C} \subseteq \left(P\big|_{\mathsf{dom}(R)} \backslash R\right) ; \mathbb{T}_{C,C}$. In the literature, this condition is stated only as sufficient, but it can be shown that it is equivalent to our condition in Lemma 2, which is necessary and sufficient. The advantage of our conditions for the definedness of demonic left and right residuals compared to the abstract relation-algebraic presentation is a better insight into the constraints that the relations R, P, and Q must satisfy in order for $R \mathbin{/\!\!/} Q$ and $P \mathbin{\backslash\!\!\backslash} R$ to be well defined.

It is worth explaining why the demonic residuals are more suitable compared to their angelic counterparts. Let us consider the relations depicted in Fig. 8. Here, if seen as specifications, the angelic residual R/Q allows the dead end (a_1, b_2) where an implementation could get stuck, whereas the demonic residual $R \mathbin{/\!\!/} Q$ does not allow any dead ends. Moreover, both demonic residuals in the figure are less restrictive than their angelic counterparts without breaking refinement: R/Q, but not $R \mathbin{/\!\!/} Q$, asks its implementations to deal with a_2, which is not an input of interest for R; similarly, $P\backslash R$, but not $P \mathbin{\backslash\!\!\backslash} R$, asks its implementations to deal with b_2.

The demonic operators have the same precedence as their angelic counterparts. For more details on the demonic calculus of relations, we refer the reader to [4–6].

3.3. Demonic factorization of relations

To answer the question whether acceptable software exists in the four-variable model, we are interested in existence conditions for the dotted arrows in the commutative diagram depicted in Fig. 9. The results will be applied to the four-variable model in the subsequent sections.

Ultimately, we are interested in necessary and sufficient conditions for the existence of a demonic factor X such that $P \mathbin{\square} X \mathbin{\square} Q \sqsubseteq R$, which we call a *demonic mid factor* of R through P and Q. Demonic composition is an associative operation [36]: $P \mathbin{\square} (X \mathbin{\square} Q) = (P \mathbin{\square} X) \mathbin{\square} Q$. The associativity of $\square$ indicates that both diagonals AC and BD are necessary for X to exist. This suggests that the existence of the diagonals might also be a sufficient condition. As it turns out, this is not the case. By applying Lemma 2 in $\Delta A, B, D$, the necessary and sufficient condition for diagonal BD to exist is:

$$\mathsf{dom}(R) \subseteq \mathsf{dom}(P) \wedge \forall b \in \mathsf{ran}\left(P\big|_{\mathsf{dom}(R)}\right) \bullet \exists d \in D \bullet \left(P\big|_{\mathsf{dom}(R)}\right)^{\smile}(b) \subseteq R^{\smile}(d)\,. \tag{13}$$

By definition, the largest relation, with respect to $\sqsubseteq$, for the diagonal BD is the demonic right residual $P \mathbin{\backslash\!\!\backslash} R$. The necessary and sufficient condition for the existence of diagonal AC is obtained by applying Lemma 1 in $\Delta A, C, D$:

$$\forall a \in \mathsf{dom}(R) \bullet \exists c \in \mathsf{dom}(Q) \bullet Q(c) \subseteq R(a)\,. \tag{14}$$

The largest relation, with respect to $\sqsubseteq$, for the diagonal AC is the demonic left residual $R \mathbin{/\!\!/} Q$.

While conditions (13) and (14) are both necessary for X, Fig. 10 provides a counterexample to the sufficiency of their conjunction. Condition (13) is satisfied because $\mathsf{dom}(R) \subseteq \mathsf{dom}(P)$ and $\left(P\big|_{\mathsf{dom}(R)}\right)^{\smile}(b_1) = \{a_1, a_2\} \subseteq R^{\smile}(d_2) = \{a_1, a_2\}$. Condition (14) is also satisfied because $Q(c_1) = \{d_1\} \subseteq R(a_1) = \{d_1, d_2\}$ and $Q(c_3) = \{d_3\} \subseteq R(a_2) = \{d_2, d_3\}$. However, if $(b_1, c_1) \in X$, then a_2 can be connected to d_1 via $P \mathbin{\square} X \mathbin{\square} Q$ although $(a_2, d_1) \notin R$; similarly, if $(b_1, c_3) \in X$, then a_1 can reach d_3 via $P \mathbin{\square} X \mathbin{\square} Q$ although $(a_1, d_3) \notin R$. Consequently, there is no relation X such that $P \mathbin{\square} X \mathbin{\square} Q \sqsubseteq R$, although both (13) and (14) are satisfied. It is only when $(c_2, d_2) \in Q$ that there is an $X = \{(b_1, c_2)\}$ such that $P \mathbin{\square} X \mathbin{\square} Q \sqsubseteq R$. It can be seen in Fig. 10 that d_2 enjoys a special property: the amount of "confusion" at the input of R to produce d_2 is at least the same as the amount of "confusion" at the input of P to produce b_1, that is, $\left(P\big|_{\mathsf{dom}(R)}\right)^{\smile}(b_1) = \{a_1, a_2\} \subseteq R^{\smile}(d_2) = \{a_1, a_2\}$. This suggests that Q reaching points similar to d_2 must be part of a necessary and sufficient condition for X to exist.

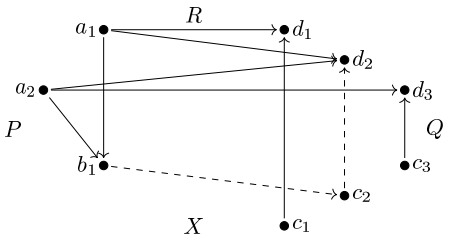


Fig. 10. The diagonals are not sufficient for a demonic mid factor.

Lemma 3. *Given three relations $R \subseteq A \times D$, $P \subseteq A \times B$, and $Q \subseteq C \times D$, there exists a demonic mid factor $X \subseteq B \times C$ such that $P \sqcap X \sqcap Q \sqsubseteq R$ if and only if the following conditions are both satisfied:*

(i) $\mathrm{dom}(R) \subseteq \mathrm{dom}(P)$;

(ii) $\forall b \in \mathrm{ran}\left(P|_{\mathrm{dom}(R)}\right) \cdot \exists c \in \mathrm{dom}(Q) \cdot Q(c) \subseteq \left\{ d \in D \;\middle|\; \left(P|_{\mathrm{dom}(R)}\right)^{\smile}(b) \subseteq R^{\smile}(d) \right\}$.

Proof. The geometrical interpretation in Fig. 10 of the associativity of $\sqcap$ is that it does not matter if we use the diagonal BD or the diagonal AC to arrive to the condition for the existence of X. As such, it suffices to use the diagonal BD and show that the conditions in Lemma 3 are necessary and sufficient for X such that $P \sqcap (X \sqcap Q) \sqsubseteq R$.

$$
\begin{aligned}
& \exists X \cdot P \sqcap (X \sqcap Q) \sqsubseteq R \\
\Leftrightarrow\ & \langle \text{by definition of } \backslash \text{ \& Lemma 2 applied in } \Delta A, B, D \rangle \\
& (\exists X \cdot X \sqcap Q \sqsubseteq P \backslash R) \wedge (13) \\
\Leftrightarrow\ & \langle \text{by Lemma 1 applied in } \Delta B, C, D \rangle \\
& (\forall b \in \mathrm{dom}(P \backslash R) \cdot \exists c \in \mathrm{dom}(Q) \cdot Q(c) \subseteq (P \backslash R)(b)) \wedge (13) \\
\Leftrightarrow\ & \left\langle \mathrm{dom}(P \backslash R) = \mathrm{ran}\left(P|_{\mathrm{dom}(R)}\right) \;\&\; (P \backslash R)(b) = \left\{ d \in D \;\middle|\; \left(P|_{\mathrm{dom}(R)}\right)^{\smile}(b) \subseteq R^{\smile}(d) \right\} \right\rangle \\
& \mathrm{dom}(R) \subseteq \mathrm{dom}(P) \\
& \wedge \forall b \in \mathrm{ran}\left(P|_{\mathrm{dom}(R)}\right) \cdot \exists c \in \mathrm{dom}(Q) \cdot Q(c) \subseteq \left\{ d \in D \;\middle|\; \left(P|_{\mathrm{dom}(R)}\right)^{\smile}(b) \subseteq R^{\smile}(d) \right\}. \qquad \square
\end{aligned}
$$

Lemma 4. *Given relations $R \subseteq A \times D$, $P \subseteq A \times B$, $X \subseteq B \times C$, and $Q \subseteq C \times D$, if $P \sqcap X \sqcap Q \sqsubseteq R$, then $X \sqsubseteq P \backslash R \mathbin{/\!\!/} Q$.*

Proof. For any X such that $P \sqcap (X \sqcap Q) \sqsubseteq R$ we have that $P \sqcap (X \sqcap Q) \sqsubseteq R \Leftrightarrow X \sqcap Q \sqsubseteq P \backslash R \Leftrightarrow X \sqsubseteq (P \backslash R) \mathbin{/\!\!/} Q$ by using the definitions of $\backslash$ and $/\!\!/$, respectively. It is also the case that for any X such that $(P \sqcap X) \sqcap Q \sqsubseteq R$ we have that $(P \sqcap X) \sqcap Q \sqsubseteq R \Leftrightarrow P \sqcap X \sqsubseteq R \mathbin{/\!\!/} Q \Leftrightarrow X \sqsubseteq P \backslash (R \mathbin{/\!\!/} Q)$. Considering that the demonic composition is associative, we drop the parentheses and say that any solution of the inequality $P \sqcap X \sqcap Q \sqsubseteq R$, if it exists, is a demonic refinement of the residual $P \backslash R \mathbin{/\!\!/} Q$. $\square$

An implication of Lemma 4 is that the residual $P \backslash R \mathbin{/\!\!/} Q$ is the largest solution, with respect to $\sqsubseteq$, of the inequation $P \sqcap X \sqcap Q \sqsubseteq R$. We call this residual the *demonic mid residual* of R by P and Q. By Lemma 3 and Lemma 4, the demonic mid residual is defined only if a demonic mid factor exists. Lemma 3 also gives us the value of the demonic mid residual:

$$
\begin{aligned}
P \backslash R \mathbin{/\!\!/} Q \succeq \Bigg\{ (b, c) \in B \times C \;\Bigg|\; & b \in \mathrm{ran}\left(P|_{\mathrm{dom}(R)}\right) \wedge c \in \mathrm{dom}(Q) \wedge \\
& Q(c) \subseteq \left\{ d \in D \;\middle|\; \left(P|_{\mathrm{dom}(R)}\right)^{\smile}(b) \subseteq R^{\smile}(d) \right\} \Bigg\}. \qquad (15)
\end{aligned}
$$

4. Implementability of system requirements

In this section we give a formal characterization for the implementability of system requirements in the four-variable model. To this end, we redefine feasibility of system requirements and acceptability of software using the demonic calculus of relations, strengthening the angelic conditions proposed by Parnas and Madey[3] and discussed in Sections 2.3 and 2.4. We also give a necessary and sufficient condition for the existence of acceptable software. Our approach also yields a formal characterization of the software requirements in terms of the least restrictive, or weakest, software specification.

4.1. Redefining feasibility of system requirements

Revisiting Fig. 3, we can see that $REQ \sqcap NAT = \{(m_1, c_1), (m_2, c_3)\}$ drops $(m1, c2)$ from REQ because this pair does not belong to NAT. As such, for the inputs in the domain of NAT, the demonic intersection of REQ with NAT retains only that part of the system requirements that is physically meaningful. This suggests that we should ask a system to implement $REQ \sqcap NAT$ instead of REQ. If we want the system requirements to specify only physically meaningful behaviors for the inputs allowed by NAT, then the following new definition should be used for the feasibility of the system requirements.

Definition 5. System requirements REQ are *feasible* with respect to a physical environment NAT if and only if $REQ = REQ \sqcap NAT$.

This notion of feasibility is stronger than the feasibility notion proposed by Parnas and Madey [3] and described in Section 2.3. The latter allows REQ to specify outputs not allowed by NAT as long as REQ also specifies at least one output allowed by NAT for every input in the domain of NAT (conditions (1) and (2)). The feasibility notion we propose in Definition 5, on the other hand, requires REQ to specify only outputs allowed by NAT for every input in the domain of NAT. This is a consequence of the following theorem, which also ensures that the demonic intersection of REQ with NAT in Definition 5 is always well defined.

Theorem 6. *System requirements REQ are feasible with respect to a physical environment NAT if and only if REQ is a demonic refinement of NAT:*

$$REQ = REQ \sqcap NAT \Leftrightarrow REQ \sqsubseteq NAT.$$

Proof. The statement of this theorem holds in a lattice for any REQ and NAT. However, because the demonic lattice is a join semi-lattice, the demonic meet $\sqcap$ is not always well defined. Thus, we need to make sure that $\sqcap$ is well defined in both directions of $\Leftrightarrow$.

For any two relations REQ and NAT, we can write REQ as

$$REQ = REQ\big|_{\mathsf{dom}(NAT)} \cup REQ\big|_{\overline{\mathsf{dom}(NAT)}}. \tag{16}$$

By the definition we also have that

$$REQ \sqcap NAT = (REQ \cap NAT) \cup REQ\big|_{\overline{\mathsf{dom}(NAT)}} \cup NAT\big|_{\overline{\mathsf{dom}(REQ)}}. \tag{17}$$

"$\Rightarrow$" direction:

Assuming $REQ = REQ \sqcap NAT$, it follows by (17) that

$$REQ = (REQ \cap NAT) \cup REQ\big|_{\overline{\mathsf{dom}(NAT)}} \cup NAT\big|_{\overline{\mathsf{dom}(REQ)}}. \tag{18}$$

By combining (16) and (18), and canceling $REQ\big|_{\overline{\mathsf{dom}(NAT)}}$ on both sides, we get

$$REQ\big|_{\mathsf{dom}(NAT)} = (REQ \cap NAT) \cup NAT\big|_{\overline{\mathsf{dom}(REQ)}}. \tag{19}$$

For this equality to hold, $NAT\big|_{\overline{\mathsf{dom}(REQ)}}$ has to be empty, which implies that

$$\mathsf{dom}\,(NAT) \subseteq \mathsf{dom}\,(REQ)\ . \tag{20}$$

With $NAT\big|_{\overline{\mathsf{dom}(REQ)}}$ empty, (19) becomes

$$REQ\big|_{\mathsf{dom}(NAT)} = REQ \cap NAT\,. \tag{21}$$

Equation (21) implies that $REQ\big|_{\mathsf{dom}(NAT)} \subseteq NAT$.

Starting from $REQ = REQ \sqcap NAT$, we have shown that $\mathsf{dom}\,(NAT) \subseteq \mathsf{dom}\,(REQ)$ and $REQ\big|_{\mathsf{dom}(NAT)} \subseteq NAT$. By the definition of demonic refinement, this means that $REQ \sqsubseteq NAT$.

We still need to make sure that the demonic intersection of REQ and NAT in $REQ = REQ \sqcap NAT$ is always well defined. Because (20) holds, we have that

$$\mathsf{dom}\,(REQ) \cap \mathsf{dom}\,(NAT) = \mathsf{dom}\,(NAT)\ . \tag{22}$$

By (21), $\mathsf{dom}\left(REQ\big|_{\mathsf{dom}(NAT)}\right) = \mathsf{dom}\,(REQ \cap NAT)$. Because (20) holds, we also have that $\mathsf{dom}\left(REQ\big|_{\mathsf{dom}(NAT)}\right) = \mathsf{dom}\,(NAT)$. Consequently:

$$\mathsf{dom}\,(REQ \cap NAT) = \mathsf{dom}\,(NAT)\ . \tag{23}$$

By (22) and (23), the following equality holds:

$$\mathsf{dom}\,(REQ \cap NAT) = \mathsf{dom}\,(REQ) \cap \mathsf{dom}\,(NATT)\ . \tag{24}$$

This equality is exactly the same as the second condition of the feasibility notion proposed by Parnas and Madey (condition (2) in Section 2.3), which is in fact equivalent to the compatibility condition (5) of *REQ* and *NAT* because $\mathsf{dom}\,(REQ \cap NAT) \subseteq \mathsf{dom}\,(REQ) \cap \mathsf{dom}\,(NAT)$ is satisfied by any *REQ* and *NAT* (i.e., it is a tautology). As such, $REQ \sqcap NAT$ is well defined if $REQ = REQ \sqcap NAT$.

"$\Leftarrow$" direction:

Assuming $REQ \sqsubseteq NAT$, we have by the definition of demonic refinement that $\mathsf{dom}\,(NAT)$ is contained in $\mathsf{dom}\,(REQ)$. This implies that

$$NAT\big|_{\overline{\mathsf{dom}(REQ)}} = \bot\!\!\!\bot\,. \tag{25}$$

Demonic refinement also implies that $REQ\big|_{\mathsf{dom}(NAT)} \subseteq NAT$. Because $REQ\big|_{\mathsf{dom}(NAT)} \subseteq REQ$ holds for any *REQ* and *NAT*, we also have that $REQ\big|_{\mathsf{dom}(NAT)} \subseteq REQ \cap NAT$. Moreover, the converse inclusion $REQ \cap NAT \subseteq REQ\big|_{\mathsf{dom}(NAT)}$ is trivially satisfied by any *REQ* and *NAT*. Therefore (21) holds. From (16), (17), (21) and (25), it follows that $REQ = REQ \sqcap NAT$.

As shown in the "$\Rightarrow$" direction, this implies that $REQ \sqcap NAT$ is always well defined. □

Because Definition 5 implies that $\mathsf{dom}\,(NAT) \subseteq \mathsf{dom}\,(REQ)$, a question arises as to what the system requirements should do about the inputs outside the domain of *NAT*. These inputs can be assumed to never happen under normal circumstances, but, for increased robustness of the system, they can be used to specify system response for the cases when the normal behavior of the environment is perturbed by some phenomena that are independent of the system.

The check for the feasibility of system requirements can be done as part of the requirements validation process. If the requirements are not feasible with respect to the environment to be controlled by the system, then no implementation will fully satisfy them. As such, feasibility is a necessary implementability condition for system requirements.

4.2. Redefining system and software acceptability

We now redefine the angelic acceptability notion of Parnas and Madey [3], described in Section 2.4, in the demonic calculus of relations.

Definition 7. A system implementation *SYS* is acceptable with respect to system requirements *REQ* and physical environment *NAT* if and only if $SYS \sqsubseteq REQ$, where *REQ* is feasible with respect to *NAT*.

Given an acceptable system implementation *SYS*, Theorem 6 ensures that the following refinement ordering holds:

$$SYS \sqsubseteq REQ = REQ \sqcap NAT \sqsubseteq NAT$$

Consequently, an acceptable system implementation will sense all the inputs that are possible from the environment and, for these inputs, will produce only outputs allowed by the physical environment. The inputs outside the domain of *NAT*, but in the domain of *REQ*, can be assumed to never happen under normal environmental circumstances; these inputs can be used for specifying fault-tolerant behavior for abnormal circumstances when the environment is perturbed by phenomena that are independent of the system. Allowing arbitrary behavior outside the domain of *REQ* should present no danger as it is assumed that, for a final product, hazard analyses have been conducted and all the inputs that could lead to hazardous system behavior have been added to the domain of *REQ* as additional safety requirements.

In Parnas and Madey [3], a system implementation is given as $SYS = IN ; SOF ; OUT$. As seen in Section 3, angelic composition allows dead ends between the composed relations and leads to the problem described in Section 2.4.3. If in the example given in Fig. 4 we redefine the system implementation using demonic composition, then $SYS = IN \,\square\, SOF \,\square\, OUT = \bot\!\!\!\bot$ and, by Definition 7, it will not demonically refine a non-empty $REQ = REQ \sqcap NAT$. Consequently, this system implementation will not be acceptable.

Considering that an acceptable software has to be part of an acceptable system implementation, we give the following definition for acceptability of software.

Definition 8. A software specification *SOF* is *acceptable* with respect to system requirements *REQ*, input interface *IN*, output interface *OUT*, and environment *NAT* if and only if $IN \,\square\, SOF \,\square\, OUT \sqsubseteq REQ$, where *REQ* is feasible with respect to *NAT*.

4.3. A necessary and sufficient implementability condition

Assuming that the system requirements are feasible with respect to the environment, the implementability of the system requirements reduces to the existence of acceptable software with respect to the input and output devices. The mathematical question we ask is, given relations *NAT*, *REQ*, *IN*, and *OUT*, does a relation *SOF* exist such that $IN \boxdot SOF \boxdot OUT \sqsubseteq REQ \parallel NAT$? The following theorem answers this question.

Theorem 9. *Given feasible system requirements REQ, input interface IN, output interface OUT, and environment NAT, there exists an acceptable software specification SOF if and only if the following conditions are both satisfied:*

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

(ii) for any software input $i \in \mathsf{ran}\left(IN|_{\mathsf{dom}(REQ \parallel NAT)}\right)$ *there exists a software output* $o \in \mathsf{dom}\,(OUT)$ *such that*

$$OUT(o) \subseteq \left\{ c \in \mathbf{C} \;\middle|\; \left(IN|_{\mathsf{dom}(REQ \parallel NAT)}\right)^{\smile}(i) \subseteq (REQ \parallel NAT)^{\smile}(c) \right\}\!\Big\}.$$

Proof. An acceptable *SOF* is a demonic mid factor of $REQ \parallel NAT$ through *IN* and *OUT* (see Section 3.3). As such, the current theorem is a direct consequence of the necessary and sufficient existence condition for a demonic mid factor given in Lemma 3. □

In Theorem 9, *IN* and *OUT* are coupled. In practical terms, this means that for the system requirements to be implementable, the input and output hardware interfaces cannot be, in general, designed independently of each other. In [37] we proved two stronger implementability conditions that decouple *IN* and *OUT*. These implementability conditions are sufficient, but not necessary.

4.4. Software requirements

The four-variable model does not explicitly specify the software requirements, but rather bounds them by specifying the system requirements and the input and output hardware interfaces of the system. The software engineers are left with the problem of how to construct software that satisfies the system requirements and input/output interfacing constraints. Extracting the software requirements from these specifications is "often an exercise in frustration" [1], hence an automated method would offer a significant advantage. In this section we give a mathematical characterization of the software requirements that offers a sound starting point for devising such a method.

From Definition 8 and Lemma 4, we have that any acceptable *SOF*, if it exists, is a demonic refinement of the demonic mid residual $IN \mathbin{\backslash\!\backslash} (REQ \parallel NAT) \mathbin{/\!\!/} OUT$. As a result, this residual is the least restrictive software specification, or the *weakest software specification*, as it leaves open most software design options. In this sense, it describes the software requirements.

Definition 10. Given feasible system requirements *REQ*, input interface *IN*, output interface *OUT*, and environment *NAT*, the software requirements SOF_{req} are given by the demonic mid residual of $REQ \parallel NAT$ through *IN* and *OUT*:

$$SOF_{req} \stackrel{\text{def}}{=} IN \mathbin{\backslash\!\backslash} (REQ \parallel NAT) \mathbin{/\!\!/} OUT\,.$$

The software requirements SOF_{req} are well defined only when an acceptable *SOF* exists, that is, when Theorem 9 is satisfied. A well defined SOF_{req} is a sound starting point for the software design process. A software design and, eventually, a program are guaranteed to be acceptable by construction if they are a demonic refinement of a well defined SOF_{req}.

If the software requirements are well defined, then they can, in principle, be derived by "calculating" the value of the residual $IN \mathbin{\backslash\!\backslash} (REQ \parallel NAT) \mathbin{/\!\!/} OUT$. One way to calculate this demonic residual is to use matrices. In general, the demonic operations are defined in terms of angelic operations, which can be calculated as operations on the adjacency matrices of the graphs associated with the relations: composition is matrix multiplication, converse is matrix transposition, etc. [26,38]. RelView,[4] with its library Kure2,[5] is a tool that supports the manipulation of relations represented as Boolean matrices using an optimized implementation based on binary decision diagrams. This approach would necessarily work only for sufficiently small, finite relations. Another way to calculate the residual $IN \mathbin{\backslash\!\backslash} (REQ \parallel NAT) \mathbin{/\!\!/} OUT$ is to use the following formula, obtained from (15):

$$SOF_{req} \succeq \left\{ (i,o) \in \mathbf{I} \times \mathbf{O} \;\middle|\; i \in \mathsf{ran}\left(IN|_{\mathsf{dom}(REQ \parallel NAT)}\right) \wedge o \in \mathsf{dom}\,(OUT) \wedge OUT(o) \subseteq \left\{ c \in \mathbf{C} \;\middle|\; \left(IN|_{\mathsf{dom}(REQ \parallel NAT)}\right)^{\smile}(i) \subseteq (REQ \parallel NAT)^{\smile}(c) \right\}\!\Big\}\,. \tag{26}$$

[4] http://www.informatik.uni-kiel.de/~progsys/relview/.
[5] http://www.informatik.uni-kiel.de/~progsys/kure2/.

From Theorem 9(ii) and (26) it follows that checking for the implementability of system requirements actually requires enumerating all the elements of SOF_{req}. This constructive nature of the necessary and sufficient implementability condition means that implementability checking also gives us the software requirements.

When calculating the software requirements is not feasible for very large relations, or in the case of infinite relations, reasoning about implementability is still possible in a higher-order logic proof assistant such as Coq, Isabelle, or PVS.

5. Example: the Pressure Sensor Trip (PST) system

In this section we analyze the implementability of the requirements for the pressure sensor trip (PST) subsystem of a nuclear reactor shutdown system, which was first described in [10]. This example highlights many of the challenges in developing such safety-critical systems, as well as the usefulness of the necessary and sufficient implementability condition given in Section 4. The implementability analysis reveals that the requirements for the PST system are not implementable given the chosen input and output devices. The hard engineering task in such situations is to find a way to make the requirements implementable. This can be done by choosing input/output devices with different capabilities, by relaxing the requirements to allow tolerances, or by doing both. Here we present a scenario in which the requirements are modified to allow tolerances and use the necessary and sufficient implementability condition to determine the tolerances needed on the requirements for the PST such that they become implementable.

5.1. Tabular specifications

We will use *tabular specifications* [39,40], or tables for short, to describe the relations in the four-variable model of the pressure sensor trip system. The reason for using tables is that they are more readily readable by humans compared to other notations used in formal specification. Depending on the kind of specifications they describe, tables with different structures as well as semantics-preserving transformations between the various types of tables have been proposed in the literature [41–47]. For the pressure sensor trip example, we will use a particular type of tables in which the characteristic predicate $R_{pred} : A \to B \to bool$ of a relation $R = \{(a, b) \in A \times B \mid R_{pred}(a, b)\}$ is described by a table with the following structure and semantics:

$cond_{1,1}(a)$	$cond_{1,2}(b)$
$\vdots$	$\vdots$
$cond_{n,1}(a)$	$cond_{n,2}(b)$

$$\begin{aligned} R_{pred}(a, b) = {} & \text{IF } cond_{1,1}(a) \text{ THEN } cond_{1,2}(b) \\ & \text{ELSEIF } \ldots \\ & \text{ELSEIF } cond_{n,1}(a) \text{ THEN } cond_{n,2}(b) \end{aligned}$$

A well defined tabular specification satisfies two properties: *disjointness* (i.e., the conditions in the first column do not overlap, otherwise logical inconsistencies might be inadvertently introduced) and *completeness* (i.e., together, the conditions in the first column cover all the possible cases so that the resulting relation is total) [11]. In general, for implementability checks we only insist on disjointness. However, the specifications of a final product must also be complete. Care must also be taken when the conditions in the second column contain conjunctions, as this can lead to logical inconsistencies when b is required to take different values at the same time.

5.2. The four-variable model of the PST

We now describe the four-variable model of the pressure sensor trip system.

5.2.1. System requirements

The PST computer is connected to a pressure sensor in the reactor. The software in the PST is required to make decisions as to whether a reactor shutdown procedure should be initiated or not. Whenever the pressure exceeds a normal operating setpoint of 2450 units, the trip computer sets its output to a "tripped" state that commands an actuator to initiate a reactor shutdown. When the pressure is less than or equal to 2400 units, the reactor should not be tripped. The requirements use a deadband region of 50 units between 2400 and 2450 to prevent "tripping" the reactor repeatedly due to sensor chatter. For pressures within the deadband region, the system is required to keep its output unchanged.

The above requirements for the PST are described formally by the following tabular specification:

$$REQ\,((pressure, PressTrip') : \mathbb{R} \times Trip,\ PressTrip : Trip) : bool =$$

$pressure \leq 2400$	$PressTrip = NotTripped$
$2400 < pressure < 2450$	$PressTrip = PressTrip'$
$2450 \leq pressure$	$PressTrip = Tripped$

Here, *REQ* is actually a function and specifies the ideal behavior expected from the system. Monitored variables are the analog voltage produced by the pressure sensor, and the previous trip state set by the software. The value of the sensor voltage is modeled by the mathematical variable *pressure* that ranges over the real numbers. The value of the previous trip state is modeled by the mathematical variable $PressTrip'$ that ranges over the set $Trip = \{Tripped, NotTripped\}$. Therefore,

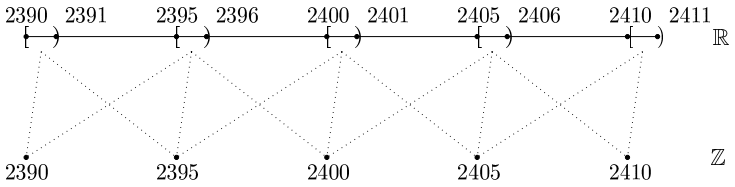


Fig. 11. Nondeterminism introduced by the ADC.

the set **M** (see Fig. 1) in the four-variable model of the PST system is the cartesian product $\mathbb{R} \times Trip$. As such, the system inputs are ordered pairs of the form $(pressure, PressTrip') \in \mathbb{R} \times Trip$. The current state of the system output is modeled by the controlled variable *PressTrip*, which, just as *PressTrip*′, ranges over the set *Trip*. Therefore, the set *Trip* of system outputs plays the role of the set **C** (see Fig. 1) in the four-variable model of the PST system.

The requirements *REQ* of the PST are assumed to be feasible with respect to the physical environment in which the PST system is to operate.

5.2.2. Input interface

The input hardware interface of the PST system consists of an analog-to-digital converter (ADC) for reading the monitored analog voltage produced by the pressure sensor. The abstraction relation *R2Z* models the functionality of the ADC and relates the monitored variable *pressure* with its corresponding software input variable, *PRES*:

$$R2Z\,(pressure : \mathbb{R},\ PRES : \mathbb{Z}) : bool = \begin{array}{|c|c|}\hline pressure \leq 0 & PRES = 0 \\ \hline 0 < pressure < 5000 & \max(0, \lfloor pressure \rfloor - 5) \leq PRES \leq \lfloor pressure \rfloor + 5 \\ \hline 5000 \leq pressure & PRES = 5000 \\ \hline \end{array}$$

The input variable *PRES* is a digital approximation of the monitored variable *pressure* that is available to the software. The effective output range of the ADC is the open integer interval (0..5000); anywhere outside this interval the output of the ADC becomes saturated. We take into account ADC inaccuracies, which are inevitable in practice. Even an ideal ADC introduces inaccuracy in the form of quantization errors (i.e., loss of accuracy due to constructing a discrete representation of a continuous quantity) [48–50]. In our example, the quantization errors are modeled by the floor function $\lfloor\,\rfloor$, which takes a real number and truncates it to its integer part. There also are inaccuracies due to hardware manufacturing tolerances, noise, etc., which manifest themselves as deviations from the actual value of the monitored pressure. For the ADC in our example, this deviation is within ± 5 units of the actual value and causes *R2Z* to be a relation, not a function. Because of these inaccuracies, the ADC introduces uncertainty (nondeterminism) in a system implementation. For example, any pressure in the real interval [2395..2406) can be mapped to the same software input $PRES = 2400$, as illustrated in Fig. 11. As we will show later, this nondeterminism causes implementability issues.

The monitored previous trip state, *PressTrip*′, is mapped to the boolean input variable *PREV* in the software via the abstraction function *Trip2Bool*:

$$Trip2Bool\,(PressTrip' : Trip,\ PREV : bool) : bool = \begin{array}{|c|c|}\hline PressTrip' = Tripped & PREV = true \\ \hline PressTrip' = NotTripped & PREV = false \\ \hline \end{array}$$

The relation *IN* in the four-variable model of the pressure sensor trip system uses the two abstractions *R2Z* and *Trip2Bool* to project the system inputs $(pressure, PressTrip') \in \mathbb{R} \times Trip$ to software inputs $(PRES, PREV) \in \mathbb{Z} \times bool$:

$$IN\,((pressure, PressTrip') : \mathbb{R} \times Trip,\ (PRES, PREV) : \mathbb{Z} \times bool) : bool = R2Z\,(pressure, PRES) \wedge Trip2Bool\,(PressTrip', PREV)$$

5.2.3. Output interface

The output interface of the pressure sensor trip system is described by the following table:

$$OUT\,(PTRIP : bool,\ PressTrip : Trip) : bool = \begin{array}{|c|c|}\hline PTRIP = true & PressTrip = Tripped \\ \hline PTRIP = false & PressTrip = NotTripped \\ \hline \end{array}$$

The software sets the boolean output variable *PTRIP* to *true* to indicate that a sensor trip has occurred and to *false* otherwise. The controlled variable *PressTrip* is then actuated accordingly by the output devices to *Tripped* or *NotTripped*. If the trip state is *Tripped*, a reactor shutdown is initiated.

The four-variable model of the PST system is depicted in Fig. 12. No environmental constraints are considered in this example. Also, the tabular specifications for *REQ*, *IN*, and *OUT* each are disjoint, complete, and internally consistent.

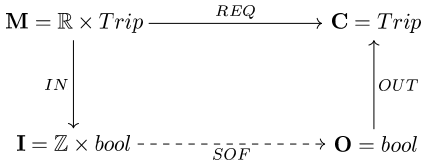


Fig. 12. The four-variable model of the PST.

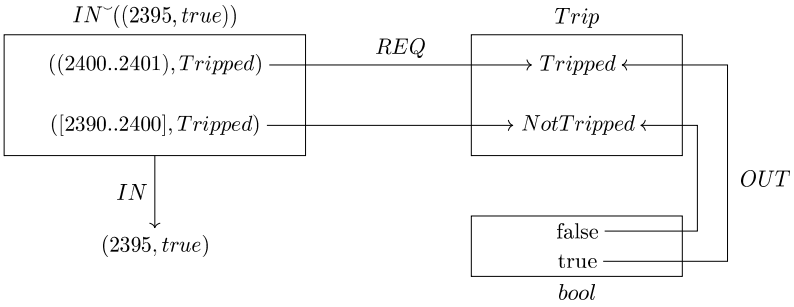


Fig. 13. Implementability issues when $(PRES, PREV) = (2395, true)$.

5.3. Implementability analysis and tolerances for the PST

Having described formally the system requirements *REQ*, input interface *IN*, and output interface *OUT* for the pressure sensor trip system, the question now is whether the system requirements are implementable or not, and, if not, what tolerances are needed on the system requirements so they become implementable. We will use the implementability condition presented in Theorem 9 to answer these two questions.

We assume that *REQ* is feasible with respect to the physical environment. Because *REQ* and *IN* are total, it is the case that $\mathsf{dom}\,(REQ) = \mathsf{dom}\,(IN)$ and $IN\big|_{\mathsf{dom}(REQ)} = IN$. Hence, by specializing Theorem 9 to this setting, we get the following necessary and sufficient implementability condition for the pressure sensor trip system:

$$\forall (PRES, PREV) \in \mathsf{ran}\,(IN) \,.\, \exists PTRIP \in \mathsf{dom}\,(OUT) \,.\, OUT(PTRIP) \subseteq \left\{ PressTrip \in Trip \;\middle|\; IN^{\smile}((PRES, PREV)) \subseteq REQ^{\smile}(PressTrip) \right\}. \tag{27}$$

There are three steps in the implementability analysis we carry out for the pressure sensor trip system. First, we find all counterexamples to (27); this will give us the largest subsets in **M** where tolerances are needed on *REQ* for some, if not all, system inputs. Second, we find which of the system inputs in the subsets identified in the first step really need tolerances and figure out the right tolerances. Formally, this means enlarging the image sets for those system inputs such that the system requirements become implementable. Usually, many solutions are possible, but a most desirable solution is one that minimally changes the requirements. Third, we derive a relaxed version of the system requirements that has the tolerances from the second step.

Step 1: Find the regions in the system input space where tolerances are needed

The universal quantifier in (27) requires checking all the software inputs, which is an infinite state space in the case of the PST system. Intuition dictates to start looking for counterexamples in the vicinities of the two setpoints specified in the system requirements. We choose to describe the analysis around the setpoint 2400 and only give the results for the analysis around the setpoint 2450.

A counterexample to (27) is found by taking $(PRES, PREV) = (2395, true)$. As seen in Fig. 11, when the software receives from the ADC the pressure approximation $PRES = 2395$, the actual pressure could have had any value in the real interval [2390..2401). Thus, $IN^{\smile}((2395, true)) = ([2390..2401), Tripped)$, with the understanding that $([2390..2401), Tripped)$ denotes all the pairs $(pressure, PressTrip') \in \mathbb{R} \times Trip$ such that $2390 \leq pressure < 2401$ and $PressTrip' = Tripped$. A problem arises because the system requirements prescribe different system responses for the pressure values in the interval [2390..2401), situation depicted in Fig. 13: on the subinterval [2390..2400], the system is asked to produce a *NotTripped* output regardless of the previous trip state, whereas on the subinterval (2400..2401) the system is asked to keep its previous trip state. For $(PRES, PREV) = (2395, true)$, the previous trip state is $PressTrip' = Tripped$. As a consequence, $IN^{\smile}((2395, true)) = ([2390..2400], Tripped) \cup ((2400..2401), Tripped) \not\subseteq REQ^{\smile}(PressTrip)$ for any $PressTrip \in Trip$. This constitutes a counterexample to (27).

If we look again at Fig. 11, it is clear that 2395 is the smallest *PRES* that can originate from actual pressures higher than the setpoint 2400. The greatest *PRES* that can originate from actual pressures less than or equal to the setpoint 2400

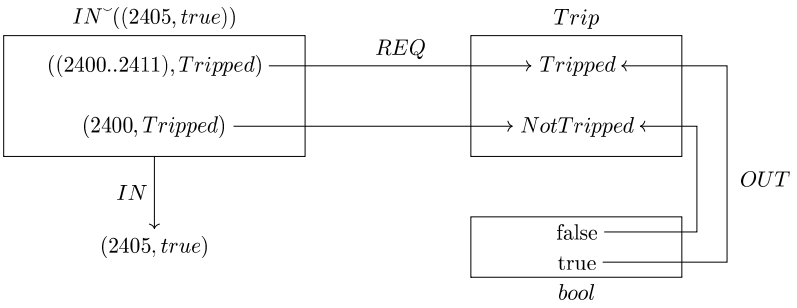


Fig. 14. Implementability issues when $(PRES, PREV) = (2405, true)$.

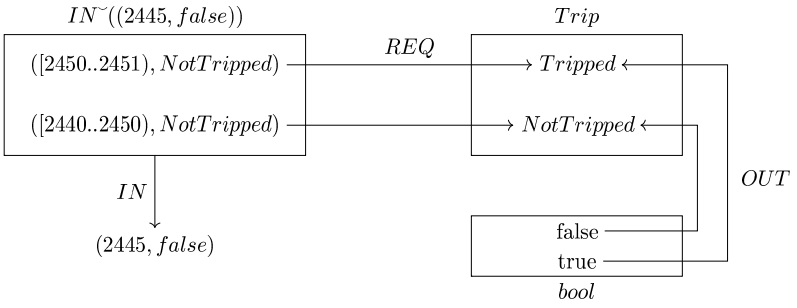


Fig. 15. Implementability issues when $(PRES, PREV) = (2445, false)$.

is 2405. The situation at $(PRES, PREV) = (2405, true)$ is illustrated in Fig. 14, where $IN^{\smile}((2405, true)) = (2400, Tripped) \cup ((2400..2411), Tripped) \nsubseteq REQ^{\smile}(PressTrip)$ for any $PressTrip \in Trip$. Consequently, $(PRES, PREV) = (2405, true)$ violates (27).

The cases when the previous trip state is $PressTrip' = NotTripped$ and $PREV = false$ are not problematic. The reason is that for pressure values less than or equal to 2400 the system requirements specify that a *NotTripped* output should be produced regardless of *PressTrip'* and that above 2400 the system output should not change. Therefore, the software inputs around the setpoint 2400 that do not satisfy (27) are the pairs $(PRES, PREV)$ such that $PRES$ is in the integer interval [2395..2405] and $PREV = true$. Consequently, the largest system input region around the setpoint 2400 where tolerances are needed is given by the pairs $(pressure, PressTrip') \in \mathbb{R} \times Trip$ such that $2390 \leq pressure < 2411$ and $PressTrip' = Tripped$.

A similar analysis around the setpoint 2450 reveals that the software inputs that do not satisfy (27) are the pairs $(PRES, PREV)$ such that $PRES$ is in the integer interval [2445..2454] and $PREV = false$. The situations at the extremities of this interval are depicted in Figs. 15 and 16. Consequently, the largest system input region around the setpoint 2450 where tolerances are needed is given by the pairs $(pressure, PressTrip') \in \mathbb{R} \times Trip$ such that $2440 \leq pressure < 2460$ and $PressTrip' = NotTripped$.

This gives us the two system input regions where there definitely are system inputs for which the system requirements need tolerances. Allowing tolerances outside these regions is completely unnecessary.

Step 2: Find proper tolerances

The second step is to figure out which system inputs in the regions found at Step 1 really need tolerances and what these tolerances are.

Usually, many solutions are possible. For the pressure sensor trip system, as can be seen in Figs. 13 and 14, we have three options for relaxing REQ around the setpoint 2400 such that the software inputs $(PRES, PREV) = ([2395..2405], true)$ will satisfy the necessary and sufficient implementability condition (27):

1. for $(pressure, PressTrip') = ([2390..2400], Tripped)$, the system requirements give an implementation the choice to set the controlled variable *PressTrip* to either *Tripped* or *NotTripped*;
2. for $(pressure, PressTrip') = ((2400..2411), Tripped)$, the system requirements give an implementation the choice to set the controlled variable *PressTrip* to either *Tripped* or *NotTripped*;
3. both previous options combined.

Around the setpoint 2450 we also have three options for relaxing REQ so that the software inputs $(PRES, PREV) = ([2445..2454], false)$ will satisfy the necessary and sufficient implementability condition (27) (see Figs. 15 and 16):

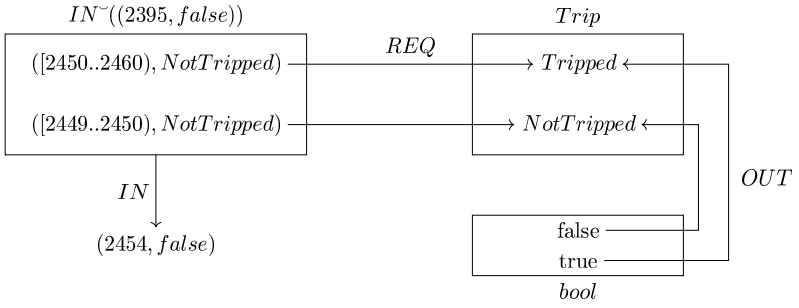


Fig. 16. Implementability issues when $(PRES, PREV) = (2454, false)$.

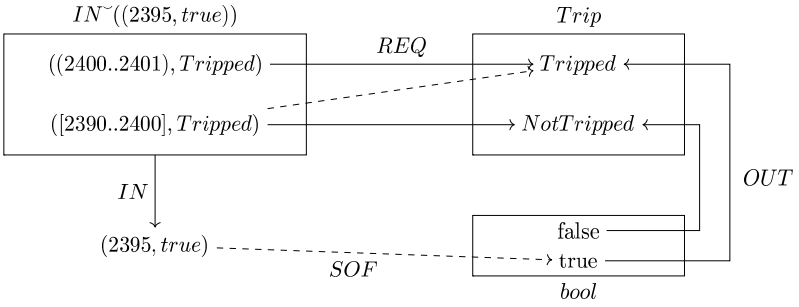


Fig. 17. The 4-variable diagram commutes when proper tolerances are allowed on system requirements.

1. for $(pressure, PressTrip') = ([2440..2450), NotTripped)$, the system requirements give an implementation the choice to set the controlled variable *PressTrip* to either *Tripped* or *NotTripped*;
2. for $(pressure, PressTrip') = ([2450..2460), NotTripped)$, the system requirements give an implementation the choice to set the controlled variable *PressTrip* to either *Tripped* or *NotTripped*;
3. both previous options combined.

Choosing one of the three tolerance options for each of the two setpoints produces a relaxed, implementable version of the initial system requirements. There are nine such possibilities. The first two tolerance options around the two setpoints are minimal changes to the system requirements. The third options would relax the requirements more than necessary.

Step 3: Derive relaxed system requirements

We now present the effect of choosing the first tolerance option for the setpoint 2400, combined with the second tolerance option for the setpoint 2450 that were described at Step 2. The other eight possibilities to relax *REQ* are not explored here, but a similar process and reasoning can be used to obtain them.

Fig. 17 depicts how the chosen tolerances make an acceptable *SOF* possible. For brevity, the figure illustrates only for $(PRES, PREV) = (2395, true)$ how the diagram of the four-variable model commutes.

The resulting system requirements with the chosen tolerances are given by the following relation REQ':

$$REQ'\,((pressure, PressTrip') : \mathbb{R} \times Trip,\ PressTrip : Trip) : bool =$$

$pressure < 2390$	$PressTrip = NotTripped$
$2390 \leq pressure \leq 2400$	$PressTrip = NotTripped$ $\vee PressTrip = PressTrip'$
$2400 < pressure < 2450$	$PressTrip = PressTrip'$
$2450 \leq pressure < 2460$	$PressTrip = PressTrip'$ $\vee PressTrip = Tripped$
$2460 \leq pressure$	$PressTrip = Tripped$

The necessary and sufficient implementability condition (27) has helped us to find the relaxed, implementable version REQ' of the original, unimplementable system requirements *REQ*. Because REQ' satisfies (27), the tolerances it allows are sufficient for implementability. These tolerances are also necessary because if we reduced the system input regions for which tolerances are allowed, then REQ' would no longer satisfy (27). In this sense, the tolerances allowed in REQ' are minimal changes to the initial requirements *REQ* that are needed for *REQ* to become implementable. In practice, REQ' would need to be revalidated by the domain experts and client to ensure that the tolerances are acceptable.

Because REQ' satisfies the necessary and sufficient implementability condition, the demonic mid residual $IN \setminus\!\!\setminus REQ \mathbin{/\!\!/} OUT$ is defined and (26) gives us the corresponding software requirements:

$$SOF_{req}\,((PRES, PREV) : \mathbb{Z} \times bool,\ PTRIP : bool) : bool = \begin{array}{|c|c|}\hline PRES < 2395 & PTRIP = false \\ \hline 2395 \leq PRES < 2455 & PTRIP = PREV \\ \hline 2455 \leq PRES & PTRIP = true \\ \hline \end{array}$$

6. Related work and discussion

A method for assessing the implementability of system requirements early in system development may save time and resources. To be implementable, the system requirements must obey the laws of the physical environment in which the system is to operate, a property called *feasibility* of system requirements. Another condition necessary for implementability is the existence of a software specification that satisfies the constraints imposed by the system requirements and chosen hardware interfaces. Such a software specification is called *acceptable*. We formalized the feasibility of system requirements and acceptability of software in the demonic calculus of relations, strengthening the angelic definitions proposed by Parnas and Madey [3], and proved a necessary and sufficient implementability condition for system requirements. The demonic approach offers guarantees of total correctness, rather than the partial correctness guarantees of an angelic approach. The demonic setting also allowed us to deal with partial specifications, which are rather the norm in early stages of system development.

The implementability results presented in the this paper are very general. The relations *REQ*, *IN*, *OUT*, and *SOF* model input–output behaviors without internal states. Also, we did not assume any structure on the sets **M**, **C**, **I**, and **O**. On one hand, this generality facilitates foundational principles for implementability in the four-variable model. On the other hand, our implementability condition does not explicitly consider constraints that a practical implementation has to deal with, such as, for example, timing. Time can be added explicitly to the four-variable model by treating the elements of **M**, **C**, **I**, and **O** as functions of time [3,10,51]. A useful research direction would be to specialize our implementability condition to include timing constraints.

Methods for assessing the existence of acceptable software in the four-variable model have not received much attention in the literature. Of the few examples, Lawford et al. [10] give, without proof, a necessary condition for the existence of *SOF* in a functional variant of the four-variable model. In the context of real-time systems, Hu et al. [52] address in a functional four-variable model the ability of a software implementation to meet continuous-time requirements, such as the detection of physical events that have been enabled for a predefined amount of time; necessary and sufficient existence conditions for SOF are given for different assumptions made about the access of the software to the time of the environment.

To be more useful in practice, our implementability check needs to be supported by tools. The necessary and sufficient condition suggests a general algorithm for checking the implementability of system requirements. We have not investigated the complexity of such an algorithm, however, developing heuristics that exploit the particularities of a specific system will very likely improve its performance. Satisfiability Modulo Theories (SMT) solving may be another direction for an automated check. However, many SMT solvers do not cope well with formulas that have existential quantifiers within the scope of universal quantifiers, as is the case with our necessary and sufficient existence condition for acceptable software. 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 a higher-order logic proof assistant such as Coq, Isabelle, or PVS, paying the price of having to do tedious and, more than often, not trivial proofs.

An acceptable *SOF* was defined as a demonic mid factor of a feasible *REQ* through *IN* and *OUT*. Because the necessary and sufficient implementability condition ensures the existence of such a demonic factor, whenever an acceptable *SOF* is possible, the software requirements, which are given by the demonic mid residual $IN \mathbin{\backslash\!\backslash} (REQ \sqcap NAT) \mathbin{/\!\!/} OUT$, are also well defined. Thus, the software requirements are obtained as a byproduct of an implementability check. This constructive nature of the implementability condition means that spending the effort to check whether acceptable software is possible is also an effort spent to derive the software requirements.

We also addressed the need for formal methods that better reflect typical engineering practices. It is often the case in practice that requirements are not implementable without appealing to tolerances. We described how our necessary and sufficient implementability condition can be used in determining the tolerances needed on the requirements of a pressure sensor trip subsystem used in the shutdown system of a nuclear reactor. Although the results of this analysis were checked with the proof assistant Coq, the analysis itself was rather a manual effort guided by the insights gained from the implementability condition. An automated method for calculating the minimal tolerances so that a set of unimplementable requirements becomes implementable would be very useful in practice. Since this may prove to be a hard problem in general, a formal characterization of common types of tolerances that are used in practice would be helpful. An example of such typical tolerances are uniform tolerances, which allow the same deviation from the ideal behavior for every input to the system. Deriving tolerances on requirements may be seen as a "glass box activity" in the "retrenchment" approach to refinement proposed by Banach et al. [53]. This is so because obtaining the relaxed requirements with tolerances is not a refinement process, but rather an activity performed before the client, domain experts, and system designers settle down for a version of the requirements that will be implemented (i.e., the "contracted model" in retrenchment parlance).

The demonic factorization results presented in the paper can be applied to essentially any system that can be modeled using a commutative diagram similar to the one of the four-variable model. For example, such commutative diagrams

appear frequently in stepwise refinement techniques where mappings between behaviors at different levels of abstraction are frequent. If the direction of a relation (or function) is reversed compared to the four-variable model, the necessary and sufficient existence condition for a demonic mid factor can still be used provided that the converse of that relation is used instead. To our knowledge, the necessary and sufficient existence condition for a demonic mid residual is a new result in relation algebra.

For increased confidence in our results, we formalized and verified the mathematical development presented in this paper, as well as the implementability analysis and tolerances of the pressure sensor trip system with the proof assistant Coq.[6] This may be a starting point towards a formal framework that offers machine support for verified system development of safety-critical systems.

Acknowledgements

The authors would like to thank Dave Parnas for clarifications about the four-variable model. Thanks also go to Wolfram Kahl who has always found time to answer their questions about relation algebra. The detailed comments by Ali Mili on the first author's doctoral dissertation are also greatly appreciated. The ideas presented in this paper form the core of that dissertation. Not the least, the authors are grateful to the anonymous referees for their suggestions and constructive criticism that helped them to improve the paper.

References

[1] S.P. Miller, A.C. Tribble, Extending the four-variable model to bridge the system-software gap, in: Proceedings of the 20th IEEE Digital Avionics Systems Conference, 2001.
[2] J. Knight, Fundamentals of Dependable Computing for Software Engineers, Innovations in Software Engineering and Software Development Series, Chapman & Hall/CRC, 2012.
[3] D.L. Parnas, J. Madey, Functional documents for computer systems, Sci. Comput. Program. 25 (1) (1995) 41–61.
[4] M. Frappier, A relational basis for program construction by parts, Ph.D. thesis, Computer Science Department, University of Ottawa, 1995.
[5] J. Desharnais, A. Mili, T. Nguyen, Refinement and demonic semantics, Ch. 11, in: Brink et al. [15], pp. 166–183.
[6] W. Kahl, Refinement and development of programs from relational specifications, in: ENTCS, Electron. Notes Theor. Comput. Sci. 44 (3) (2003) 51–93.
[7] L.M. Patcas, M. Lawford, T. Maibaum, From system requirements to software requirements in the four-variable model, in: S. Schneider, H. Treharne, T. Margaria, J. Padberg, G. Taentzer (Eds.), Proceedings of the Automated Verification of Critical Systems (AVoCS) 2013, in: Electronic Communications of the EASST, vol. 66, 2014.
[8] A. Van Schouwen, 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.
[9] S. Faulk, J. Finneran, J. Kirby, S. Shash, J. Sutton, Experience applying the CoRE method to the Lockhead C-130J software requirements, in: Ninth Annual Conference on Computer Assurance, Gaithersburg, Maryland, 1994.
[10] M. Lawford, J. McDougall, P. Froebel, G. Moum, Practical application of functional and relational methods for the specification and verification of safety critical software, in: Proceedings of Algebraic Methodology and Software Technology, AMAST, in: Lecture Notes in Computer Science, vol. 1816, Springer, 2000, pp. 73–88.
[11] A. Wassyng, M. Lawford, Lessons learned from a successful implementation of formal methods in an industrial project, in: K. Araki, S. Gnesi, D. Mandrioli (Eds.), FME 2003, in: Lecture Notes in Computer Science, vol. 2805, Springer, 2003, pp. 133–153.
[12] A. Wassyng, M. Lawford, Software tools for safety-critical software development, Int. J. Softw. Tools Technol. Transf. 8 (4–5) (2006) 337–354.
[13] D.L. Lempia, S.P. Miller, Requirements engineering management handbook, Tech. rep. DOT/FAA/AR-08/32, U.S. Department of Transportation, Federal Aviation Administration, June 2009.
[14] C.A. Gunter, E.L. Gunter, M. Jackson, P. Zave, A reference model for requirements and specifications, IEEE Softw. 17 (3) (2000) 37–43.
[15] C. Brink, W. Kahl, G. Schmidt (Eds.), Relational Methods in Computer Science, Advances in Computing, Springer, 1997.
[16] R.-J. Back, On correct refinement of programs, J. Comput. Syst. Sci. 23 (1) (1981) 49–68.
[17] J.M. Morris, A theoretical basis for stepwise refinement and the programming calculus, Sci. Comput. Program. 9 (3) (1987) 287–306.
[18] C. Morgan, K. Robinson, Specification statements and refinement, IBM J. Res. Dev. 31 (5) (1987) 546–555.
[19] R.-J. Back, J. von Wright, Combining angels, demons, and miracles in program specifications, Theor. Comput. Sci. 100 (2) (1992) 365–383.
[20] R.D. Maddux, Relation-algebraic semantics, Theor. Comput. Sci. 160 (1–2) (1996) 1–85.
[21] R. Berghammer, H. Zierer, Relational algebraic semantics of deterministic and nondeterministic programs, Theor. Comput. Sci. 43 (1986) 123–147.
[22] C.A.R. Hoare, J. He, The weakest prespecification, Fundam. Inform. 9 (1) (1986), (Part I) 51–84, (Part II) 217–252.
[23] S. Demri, E. Orlowska, Logical analysis of demonic nondeterministic programs, Theor. Comput. Sci. 166 (1–2) (1996) 173–202.
[24] C.A.R. Hoare, J. He, The weakest prespecification, Inf. Process. Lett. 24 (2) (1987) 127–132.
[25] C.A.R. Hoare, I.J. Hayes, H. Jifeng, C.C. Morgan, A.W. Roscoe, J.W. Sanders, I.H. Sorensen, J.M. Spivey, B.A. Sufrin, Laws of programming, Commun. ACM 30 (8) (1987) 672–686.
[26] G. Schmidt, T. Ströhlein, Relations and Graphs, Discrete Mathematics for Computer Scientists, EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1993.
[27] J. Thompson, M. Heimdahl, S.P. Miller, Specification-based prototyping for embedded systems, in: O. Nierstrasz, M. Lemoine (Eds.), Seventh ACM SIGSOFT Symposium on the Foundations of Software Engineering, FSE, in: Lecture Notes in Computer Science, vol. 1687, Springer, 1999, pp. 163–179.
[28] J. Thompson, M. Whalen, M. Heimdahl, Requirements capture and evaluation in NIMBUS: the light-control case study, J. Univers. Comput. Sci. 6 (7) (2000) 731–757.
[29] M. Heimdahl, J. Thompson, Specification based prototyping of control systems, in: Proceedings of the 19th IEEE Digital Avionics Systems Conference, 2000.
[30] A. Mili, J. Desharnais, F. Mili, Relational heuristics for the design of deterministic programs, Acta Inform. 24 (3) (1987) 239–276.
[31] A. Mili, A relational approach to the design of deterministic programs, Acta Inform. 20 (4) (1983) 315–328.
[32] N. Boudriga, F. Elloumi, A. Mili, On the lattice of specifications: applications to a specification methodology, Form. Asp. Comput. 4 (6) (1992) 544–571.

[6] Coq formalization and proofs with detailed explanations are available at www.cas.mcmaster.ca/~patcaslm/papers/2014-SCP/coq.

[33] J. Desharnais, N. Belkhiter, S.B.M. Sghaier, F. Tchier, A. Jaoua, A. Mili, N. Zaguia, Embedding a demonic semilattice in a relation algebra, Theor. Comput. Sci. 149 (2) (1995) 333–360.
[34] M. Frappier, A. Mili, J. Desharnais, A relational calculus for program construction by parts, Sci. Comput. Program. 26 (3) (1996) 237–254.
[35] J. Desharnais, A. Jaoua, F. Mili, N. Boudriga, A. Mili, A relational division operator: the conjugate kernel, Theor. Comput. Sci. 114 (2) (1993) 247–272.
[36] R.C. Backhouse, J. van der Woude, Demonic operators and monotype factors, Math. Struct. Comput. Sci. 3 (4) (1993) 417–433.
[37] L.M. Patcas, M. Lawford, T. Maibaum, A separation principle for embedded system interfacing, in: E. Albert, E. Sekerinski (Eds.), Integrated Formal Methods, iFM, in: Lecture Notes in Computer Science, vol. 8739, Springer, 2014, pp. 373–388.
[38] G. Schmidt, Relational Mathematics, Encyclopedia of Mathematics and its Applications, Cambridge University Press, 2011.
[39] D.L. Parnas, Tabular representation of relations, Tech. rep. CRL 260, McMaster University, Communications Research Laboratory, 1992.
[40] D.L. Parnas, The tabular method for relational documentation, in: RelMiS 2001, Relational Methods in Software (A Satellite Event of ETAPS 2001), in: Electronic Notes in Theoretical Computer Science, vol. 44, Elsevier Science Inc., 2003, pp. 1–26.
[41] D.L. Parnas, J. Madey, M. Iglewski, Precise documentation of well-structured programs, IEEE Trans. Softw. Eng. 20 (12) (1994) 948–976.
[42] R. Janicki, D.L. Parnas, J. Zucker, Tabular representations in relational documents, Ch. 12, in: Brink et al. [15], pp. 184–196.
[43] R. Janicki, Towards a formal semantics of Parnas tables, in: Proceedings of the International Conference on Software Engineering, ICSE, 1995, pp. 231–240.
[44] R. Janicki, R. Khedri, On a formal semantics of tabular expressions, Sci. Comput. Program. 39 (2001) 189–213.
[45] H. Shen, J. Zucker, D.L. Parnas, Table transformation tools: why and how, in: Proceedings of the 11th Conference on Computer Assurance, COMPASS, 1996, pp. 3–11.
[46] J. Zucker, Transformations of normal and inverted function tables, Form. Asp. Comput. 8 (6) (1996) 679–705.
[47] W. Kahl, Compositional syntax and semantics of tables, Software Quality Research Laboratory (SQRL) 15, Department of Computing and Software, McMaster University, 2003.
[48] M.S. Santina, A.R. Stubberud, G.H. Hostetter, Quantization effects, in: W.S. Levine (Ed.), The Control Handbook, CRC Press, IEEE Press, 1996, pp. 301–311, Ch. 15.
[49] R.H. Walden, Analog-to-digital converter survey and analysis, IEEE J. Sel. Areas Commun. 17 (4) (1999) 539–550.
[50] W. Kester (Ed.), The Data Conversion Handbook, Newnes, 2005.
[51] D.K. Peters, Deriving real-time monitors from system requirements documentation, Ph.D. thesis, McMaster University, January 2000.
[52] X. Hu, M. Lawford, A. Wassyng, Formal verification of the implementability of timing requirements, in: Formal Methods for Industrial Critical Systems, in: Lecture Notes in Computer Science, vol. 5596, Springer, 2009, pp. 119–134.
[53] R. Banach, M. Poppleton, C. Jeske, S. Stepney, Engineering and theoretical underpinnings of retrenchment, Sci. Comput. Program. 67 (2–3) (2007) 301–329.