Implementability of Requirements for Safety-Critical Embedded Systems

Implementability of Requirements for Safety-Critical Embedded Systems

By

Lucian M. Patcas, B.Eng., M.Sc.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements
for the Degree of
Doctor of Philosophy

McMaster University

DOCTOR OF PHILOSOPHY (2014) MCMASTER UNIVERSITY
(Software Engineering) Hamilton, Ontario, Canada

TITLE: Implementability of Requirements for Safety-Critical Embedded Systems

AUTHOR: Lucian M. Patcas
B.Eng., Politehnica University of Timisoara, Romania
M.Sc., University College Dublin UCD, Ireland

SUPERVISORS: Dr. Mark S. Lawford and Dr. Thomas S. E. Maibaum

NUMBER OF PAGES: xiv, 126

For Dana and Ruxandra

Abstract

Computer systems are used for controlling physical processes in many safety-critical applications. These systems are embedded into the larger system of the application and are interfaced with their physical environment using input hardware (sensors, analog-to-digital converters) and output hardware (digital to analog converters, actuators). A challenging task in designing such systems is finding the right combination of input hardware, output hardware, and software such that their integration produces systems that satisfy their requirements.

In this thesis we propose a mathematical basis for checking, without the need for developing and verifying a detailed implementation, if acceptable software exists given the chosen hardware interfaces. The requirements framework we use is the relational four-variable model proposed by Parnas and Madey. This model helps to clarify the behaviour of, and the boundaries between, the system's physical environment, hardware interfaces, and software, which are all described as input-output relations without detail about internal state.

The semantics of the four-variable model proposed by Parnas and Madey, which may be seen in relation algebraic terms as an angelic semantics, allows system descriptions that are not completely consistent with the natural laws of the physical environment. To address this issue, we redefine in the demonic calculus of relations the notion of feasibility of system requirements proposed by Parnas and Madey such that the system requirements specify for every input possible in the environment only behaviours allowed by the environment. We also redefine in the demonic calculus of relations the system and software acceptability criterion of Parnas and Madey to reject nonterminating or empty implementations, and prove a necessary and sufficient existence condition for

acceptable software. This condition has a constructive flavour and yields the weakest (least restrictive, or least refined) specification of the software requirements.

A practical implication of the necessary and sufficient condition is that in a relational four-variable model, as opposed to the functional case, the input and output hardware interfaces are mutually dependent and changes to either may require changes to the other. We prove two stronger conditions that allow the decoupling of the hardware interfaces while still guaranteeing the ability of the software to meet the system requirements.

For the cases when the system requirements are feasible, but an acceptable implementation does not exist, a typical engineering approach is to relax the requirements by allowing tolerances. We show how the necessary and sufficient condition can be used in the derivation of tolerances on the requirements for a pressure sensor trip in the shutdown system of a nuclear reactor such that the requirements become implementable.

Acknowledgements

I sincerely thank my supervisors, Prof. Mark Lawford and Prof. Tom Maibaum for their guidance and extraordinary patience they have shown me over the years. With their rare ability to grasp theoretical as well as practical ideas, Mark and Tom have helped me shape the thesis so that it has a firm theoretical basis as well as practical applicability. I am also grateful to Mark and Tom for their willingness to allow me to explore a research topic of my own choosing. It took me longer to settle down with a topic this way, but I feel that it has made me better equipped for doing research in the "real life". Not the least, Mark and Tom have created a relaxed atmosphere where I could always speak my mind.

I am also appreciative of my two other supervisory committee members, Dr. Ridha Khedri and Dr. Wolfram Kahl who have always found time to answer my questions. Their positive attitude and constant encouragement have been of a tremendous help.

I am very much in debt to Dr. David Parnas for answering my email messages and for clarifications on the four-variable model. His unique perspective on software engineering and formal methods has helped me shape the ideas presented in Chapter 4.

Dr. William Farmer has been influential in my decision to use a theorem prover to formalize and verify the mathematics presented in the thesis. His graduate lectures on practical applications of logic to software engineering and computer science have been a delight.

The financial support received from the National Sciences and Research Council of Canada (NSERC), McMaster University and the Computing and Software Department, as well as from my supervisors is gratefully acknowl-

edged.

Thanks also go to fellow graduate students Vera Pantelic, Alex Korobkine, Mark Pavlidis in the early days, and Linna Pang. They have made this endeavour a less lonely experience.

I would also like to thank my parents and my sister for everything they have done for me. They have always encouraged me and been there for me in too many ways to mention here.

Finally, my love and gratitude go to my dear wife, Dana, and our lovely daughter, Ruxandra. Dana has offered me her unconditioned love and provided for our family over the years I have been a PhD student. Without Dana's unending support this work would have not been possible. Even though Ruxandra is six years old, she has always understood when I was busy working on my research and did not have as much time to play with her as we both would have liked. When that was the case, she would come to my desk and ask if she could help me with mathematics; she would then scribble in my notes and say "See, Daddy, I can do mathematics too". It is not unlikely that some of the figures that made it into the dissertation were inspired by her doodles.

Contents

List of Figures

Chapter 1

Introduction

For any system to be built it is worth asking the question whether an implementation that satisfies the requirements is possible given the constraints imposed by the environment in which the system is to operate as well as constraints imposed by design decisions. If such an implementation is possible, then the requirements are said to be *implementable* with respect to those constraints. In this thesis we ask the question of implementability in the context of safety-critical embedded systems whose requirements and high-level design specifications are given using the four-variable model proposed in (Parnas and Madey, 1995). Nevertheless, the approach and results described in the thesis have broader implications and applicability.

1.1 A Question of Implementability

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

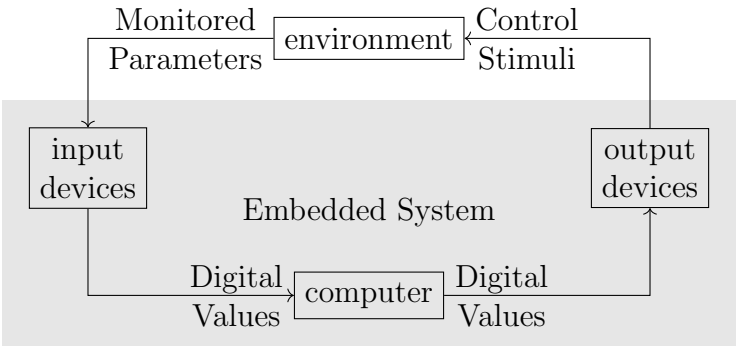


Figure 1.1: A general view of an embedded system

using input devices (sensors, analog-to-digital converters) and output devices (digital-to-analog converters, actuators), as illustrated in Figure 1.1. For reasons of flexibility and cost, the functionality required of these systems is typically implemented in software (Parnas et al., 1990; Knight, 2012). 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 task in designing these systems is finding the right combination of input devices, output devices, and software such that their integration produces a system implementation 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 implementation such that the required behaviour of the system is satisfied. Following (Parnas and Madey, 1995), we call such software *acceptable*. 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 (Miller and Tribble, 2001), (Knight, 2012, Section 2.6.3). What if no acceptable software is possible given the constraints imposed by the chosen hardware interfaces? Time and resources will be spent trying to develop and verify repeatedly a system that can never satisfy the requirements. Acceptable software is also not possible if the required behaviour

of the system is not allowed by the physical laws of the environment.

Hence, we pose the following question that systems and software engineers need to ask themselves before investing resources in developing and verifying a detailed system implementation:

Given a physical environment and a particular choice of hardware interfacing between the system and that environment, is acceptable software possible?

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 design decisions regarding the input and output devices. 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 acceptable software to become possible. Such a bidirectional interaction between systems engineering and software engineering is stressed in (Knight, 2012, Section 1.2) as being essential in producing dependable software-controlled systems.

In this thesis we propose a mathematical basis to answer the question posed above.

1.2 The Four-Variable Model

We use the four-variable model proposed in (Parnas and Madey, 1995) and illustrated in Figure 1.2. This model helps to clarify the behaviour of, and the boundaries between, the environment, hardware interfaces, and control software of an embedded system.

The four-variable 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 (Van Schouwen, 1990).

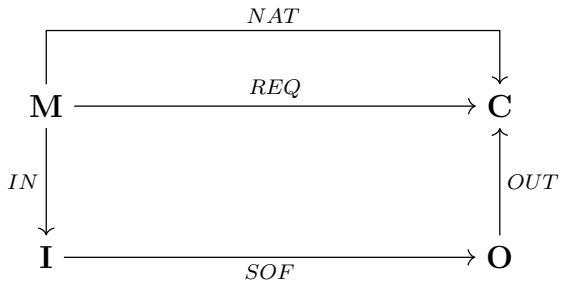


Figure 1.2: The four-variable model

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 (Faulk et al., 1994). 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 (Lawford et al., 2000; Wassyng and Lawford, 2003; Wassyng and Lawford, 2006). In 2009, the four-variable model was used extensively in the *Requirements Engineering Handbook* (Lempia and Miller, 2009) that was put together at the request of the U.S. Federal Aviation Administration.

1.2.1 System Requirements

In the four-variable model, *REQ* models a specification of 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 reactor shutdown; whenever the temperature or pressure reach abnormal values, the alarms go off and the shutdown procedure is initiated.

The sets of possible values for the monitored and controlled variables are denoted by $\mathbf{M}$ and $\mathbf{C}$, respectively. The sets $\mathbf{M}$ and $\mathbf{C}$ are not necessarily disjoint. In the four-variable model, environmental quantities are usually

modelled as functions of time (i.e., functions of a single real valued variable), especially in the context of real-time systems (Parnas and Madey, 1995; Lawford et al., 2000; Peters, 2000). In contrast, in this thesis we do not assume any structure on the sets $\mathbf{M}$ and $\mathbf{C}$. All the possible values that a monitored or controlled variable can take are members of $\mathbf{M}$ or, respectively, $\mathbf{C}$. This allows us to derive more general results that can be specialized to deal explicitly with time by considering the members of $\mathbf{M}$ and $\mathbf{C}$ as functions of time.

In this thesis, we take the system requirements specification REQ to describe the functionality required from the system. In software engineering such requirements go by the name *functional requirements*, while other types of requirements such as performance and resource utilization are called non-functional. In the sequel we will use the term "functional" with its mathematical, rather than with its software engineering meaning.

1.2.2 Environmental Constraints

The environmental constraints on the system are described by NAT (from "nature"). These constraints are due to the physical laws of the environment and are independent of the system to be built (Peters, 2000; Miller and Tribble, 2001). As such, NAT contains exactly those pairs of values of monitored and controlled variables that are possible in the environment.

Examples of environmental constraints include: the maximum rate of climb of an aircraft in the case of an avionics system (Miller and Tribble, 2001); environmental quantities that are related to each other in a certain way (e.g., temperature and pressure in a closed container) (Peters, 2000); or, events that are not physically able to occur simultaneously (Peters, 2000).

1.2.3 System Design

The possible *system behaviours* are modelled by a sequential composition of IN, SOF, and OUT. Here, IN models the functionality of the input devices 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 then converted via an analog-to-digital converter to integer values stored in a register accesible to the software via an input variable. The functionality of the output devices is modelled by OUT, which relates values of *output variables* in the software to values of controlled variables in the environment. 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. Relating values of input variables to values of output variables is SOF, which models the behaviour of the *software*, including the input/output device drivers.

The sets of the possible values of the input and output variables are denoted by $\mathbf{I}$ and $\mathbf{O}$, respectively. As was the case with the monitored and controlled variables, we do not assume any structure on $\mathbf{I}$ and $\mathbf{O}$ although the input and output variables in the four-variable model are typically treated as functions of time.

1.2.4 The Need for a Relational Framework

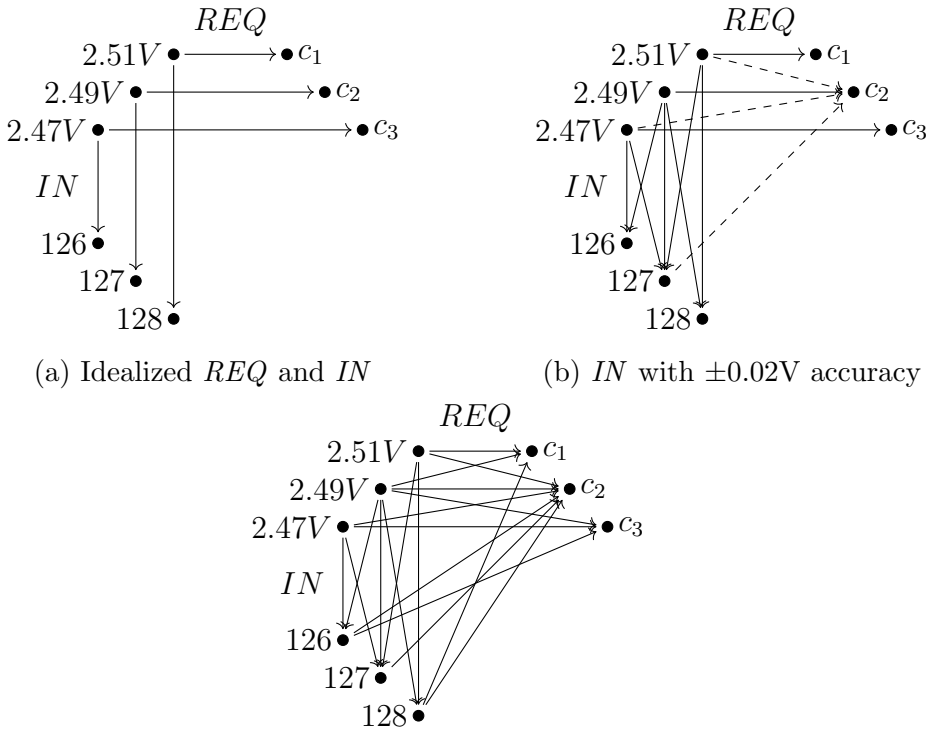
The four-variable model is in general relational, not functional. By this we mean that REQ, NAT, IN, OUT, and SOF are typically mathematical relations, not functions (Parnas and Madey, 1995; Lawford et al., 2000; Peters, 2000). This is mainly due to measurement errors in sensors and quantization errors (i.e., the difference between an analog value and its digital approximation) in analog-to-digital and digital-to-analog converters (Santina et al., 1996a; Santina et al., 1996b; Walden, 1999; Kester, 2005). Considering that IN and OUT describe the combined functionality of multiple devices, we use the term *accuracy* to describe their combined errors. The limited accuracy of the hardware interfacing induces uncertainty (nondeterminism) in a system implementation. Relations are natural candidates for modelling nondeterministic behaviours (Brink et al., 1997). In contrast, functions model deterministic behaviours. The nondeterminism of a system implementation should be accounted for by allowing tolerances on the system requirements. With tolerances, the system requirements become relational and allow a number of

acceptable system responses for the same value of a monitored variable.

For example, let us consider an input interface *IN* that models an 8-bit resolution analog-to-digital converter (ADC) which converts monitored voltages m in the range 0–5V into software input values i according to the formula $i = \lfloor m * 2^8/5 \rfloor$. Figure 1.3a 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 behaviours. If the ADC has a ± 0.02V accuracy (Figure 1.3b), then *IN* effectively 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 via *SOF* and *OUT* 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 and multiple (deterministic) system implementations become possible (Figure 1.3c). 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 implementations of the control software, then *SOF* will typically have to be a relation too. 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, represented by *NAT*, usually are relational as well. As extreme examples, if everything is possibile 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 reader should note that the relations *IN*, *OUT*, and *SOF* can model specifications of intended behaviours as well as descriptions of actual behaviours (Parnas, 2003). The distinction between these two viewpoints is often blurred since we use relations to model both specifications and actual implementations. There are situations when this distinction is nevertheless important.



(a) Idealized *REQ* and *IN*

(b) *IN* with ±0.02V accuracy

(c) *REQ* with tolerances, *IN* with ±0.02V accuracy

Figure 1.3: Motivational example for a relational four-variable model

1.2.5 Implementability in the Four-Variable Model

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 then determine *SOF* and verify whether it is acceptable with respect to *NAT*, *REQ*, *IN*, and *OUT*.

In this thesis, we allow the relations of the four-variable model to be *partial*. The rationale for allowing partial relations is that in practice, particularly in the early stages of system development, formulating specifications that deal with all the possible cases that can arise 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 (Thompson et al., 1999; Thompson et al., 2000; Heimdahl and Thompson, 2000), (Knight, 2012, Section 2.6.3). 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 part of an specification, we will not be able to fully meet a more complete version of that specification.

Our use of partial relations aligns well with the "lightweight formal methods" approach advocated by Daniel Jackson and Jeannette Wing, who argue that formal methods should focus on rapid detection of faults rather than on full proofs of correctness (Saiedian et al., 1996). In this approach, different aspects of a system are modelled and analyzed iteratively instead of trying to model the system all at once and prove that it is free of faults, the latter approach being infeasible for most practical systems. A case study of applying lightweight formal methods is presented in (Easterbrook and Callahan, 1998), where partial specifications of critical software for the International Space Station were analyzed formally and errors fixed before attempting to analyze more complete specifications. The authors concluded that more errors were found and fixed that way, and valuable insight about the system to be built was gained in the process than if a full proof of correctness had been attempted.

Thus the question of implementability of system requirements posed at the beginning of the chapter can be rephrased in the four-variable model as follows:

Given system requirements REQ, physical environment NAT, and hardware interfaces IN and OUT, all as partial relations, does an acceptable SOF exist?

It is precisely this question that we will address in this dissertation.

1.3 Related Work

In this section we outline research that is most closely related to the main question of the thesis. We have hinted thus far that we will work with specifications modelled as partial relations. When composing partial relations it can happen that the range of a relation is not completely contained in the domain of the relation with which it is being composed. Consequently, there are inputs for which the resulting composition sometimes produces expected results and some other times gets stuck in between the relations being composed. In Section 1.3.1 we describe two approaches from the area of formal program specification and semantics to deal with such nondeterministic behaviours. We then present in Section 1.3.2 a requirements framework that has been proposed in the literature as an alternative to the four-variable model. Finally, work towards necessary and sufficient existence conditions for an acceptable *SOF* is described in Section 1.3.3 in a functional setting of the four-variable model.

1.3.1 Angelic and Demonic Nondeterminism

The formal treatment of nondeterministic specifications and sequential programs originates with Dijkstra's language of guarded commands and its formal semantics, the weakest-precondition calculus (Dijkstra, 1975; Dijkstra, 1976). In this calculus, each statement of a program has two predicates associated with it: the *postcondition predicate*, which denotes the set of states that the

program can be in after the execution of the statement; and the *weakest precondition* predicate, which denotes that largest set of states that the program can be in such that after the execution of the statement the program will be in a state that satisfies the postcondition predicate. The weakest precondition predicate of a program statement is obtained from the postcondition predicate of that statement via a function called the weakest-precondition *predicate transformer*. Since a program is a sequential composition of multiple statements, the predicate transformer of a program is obtained by composing the predicate transformers of the program's statements. Mismatches between the postcondition of a program statement and the weakest precondition of the next program statement in this composition are possible and this leads to nondeterministic programs which, for the same input, sometimes produce expected results and some other times do not produce any results. Various ways to deal with such nondeterministic programs have been studied in variations of the weakest-precondition calculus, such as, for example, those proposed in (Back, 1981), (Morris, 1987), (Morgan and Robinson, 1987), (Back and von Wright, 1992), (Maddux, 1996), (Morgan, 1998), or (Back and von Wright, 1998). The two main approaches are:

- *angelic*: "in order for a computation to be successful it is enough that there exists a possible successful execution path" (Back and von Wright, 1992); and,

- *demonic*: "in order for a computation to be successful, all possible execution paths must lead to a successful result" (Back and von Wright, 1992).

The difference between the angelic and demonic approaches is, perhaps, best explained using the following metaphor. In an angelic semantics, an angel always makes the best possible choice such that the program terminates; thus, if termination is possible, the angel will ensure termination. In a demonic semantics, a demon always tries to make the worst possible choice such that the program does not terminate; thus, if nontermination is possible, the demon will find it. Because in a angelic semantics nontermination is allowed for an

input as long as termination is also possible for that input, the angelic approach guarantees only *partial correctness* of programs. In contrast, since a demonic semantics will always reveal the possibility of nontermination, the demonic approach can be used to ensure *total correctness*.

Because predicates describe subsets, predicate transformers are functions between subsets of the state space of a program. Equivalently, predicate transformers can be seen as relations on the program's state space. Relational methods for formal program specification and development were strongly advocated by Tony Hoare's group at Oxford in the mid 1980's (Hoare and He, 1985; Hoare and He, 1986; Hoare and He, 1987; Hoare et al., 1987). These methods originate from the calculus of relations started in algebraic logic by Augustus de Morgan, Charles S. Peirce, and Ernst Schröder in the second half of the nineteenth century, and revived by (Tarski, 1941) in the twentieth century. In a typical relational method, specifications and programs are thought of as input-output relations over some state space. Similarly to the approaches based on predicate transformers, the relation-algebraic approaches have to make the same decision on how they deal with the nondeterminism that arises when composing specifications (programs) into larger specifications (programs). Relation-algebraic approaches to angelic and demonic nondeterminism are described in, for example, (Berghammer and Zierer, 1986), (Hoare and He, 1986), (Frappier, 1995; Frappier et al., 1996), (Demri and Orlowska, 1996), (Maddux, 1996), (Desharnais et al., 1997), (Kahl, 2003b).

When developing safety-critical systems it is always wise to plan for all possibilities, including the "worst case" scenario. As discussed in the previous sections, nondeterministic system implementations due to hardware inaccuracies, as well as partial specifications, are all facts of life. Since a safety-critical application must always produce expected results, the partial correctness guarantees of an angelic approach are not satisfactory. Thus, we adopt a demonic approach. Also, we adopt a relation-algebraic approach rather than one based on predicate transformers. The latter were developed specifically for reasoning about the behaviour of common imperative programming constructs such as loops, conditionals, and sequential composition. Since in the four-variable model we deal with sequential compositions of nondeterministic specifications

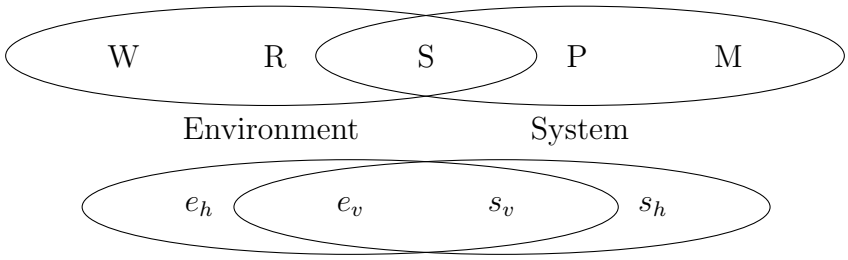


Figure 1.4: The WRSPM reference model for requirements

without internal details, a relational method feels more natural. In particular, we use the demonic calculus of relations (Frappier, 1995; Frappier et al., 1996; Desharnais et al., 1997; Kahl, 2003b).

1.3.2 Other Requirements Frameworks

The work by (Jackson and Zave, 1993; Jackson and Zave, 1995; Zave and Jackson, 1997) describes a requirements framework that shares many ideas with the four-variable model, in particular the division of phenomena (i.e., states, events) into environmental phenomena and system phenomena, as well as the formulation of system requirements only in terms of phenomena in the environment. This line of work became known as the *WRSPM reference model for requirements and specifications* (Gunter et al., 2000).

In this model, illustrated in Figure 1.4, W stands for "world" and models the environment, or application domain. The system requirements are modelled by R and represent what the customer needs from the system. The system requirements are refined into a specification S which describes the intended behaviour of the system in order to satisfy the requirements. An implementation of the intended behaviour is modelled by a program P that runs on a machine M.

The artifacts W, R, S, P, and M are described in various languages, each based on its own vocabulary. To express the relationships between the artifacts, however, their descriptions are modelled in a common language. In this language, the phenomena controlled by the environment are denoted by the set e. Some of these phenomena are visible to the system, denoted by the subset

e_v, while the other phenomena in e are hidden from the system and are denoted by the subset e_h. Thus, $e = e_h \cup e_v$. Similarly, the phenomena controlled by the system are denoted by the set s and decomposed into the subsets s_v of phenomena visible to the environment and s_h of phenomena hidden from the environment. Of course, $s = s_h \cup s_v$. Also, the sets e and s are disjoint. The phenomena in e_h, e_v, and s_v are visible to the environment and, therefore, are used to describe W and R. The phenomena in e_v, s_v, and s_h are visible to the system and used to describe P and M. The phenomena visible to both the environment and the system are used to describe the specification $S = e_v \cup s_v$. Of the papers mentioned above, (Jackson and Zave, 1995) use a complete example to explain how to classify the phenomena into environmental, system, or shared.

Although they have many commonalities, the correspondence between the WRSPM model and the four-variable model is not completely determined. The monitored variables in **M** in the four-variable model correspond to phenomena e_v in the WRSPM model, while controlled variables in **C** correspond to phenomena s_v. Thus *REQ* corresponds to S and *NAT* corresponds to W, but are more restricted than S and, respectively, W since they are expressed only in terms of those environmental phenomena that are visible to the system. The input variables in **I** and output variables in **O** correspond to hidden system phenomena e_h. Consequently, *SOF* corresponds to P, and *IN* and *OUT* belong to M. However, *IN* and *OUT* are more restricted than M, being limited to sensing and actuating.

The WRSPM model is more flexible at describing the environment and system requirements since in the four-variable model there are no environmental phenomena that are not visible to the system. Hidden environmental phenomena are useful though. For example, the system requirements may include additional functionality planned for future versions of the system. Such functionality will not be part of the requirements specification for the current release, but knowing about it will allow system designers to plan upfront. In the four-variable model there is no clear distinction between requirements and a specification of the requirements. The WRSPM model makes this distinction explicit.

On the other hand, the four-variable model clarifies the boundaries between software and other system components more cleanly than the WRSPM model. In the four-variable model, the software is decoupled from the rest of the system by using the input and output variables. In the WRSPM model, the software and the hardware interfaces are all described in terms of hidden system phenomena. Since the question of implementability we ask in this thesis has direct implications on the interaction between software engineering and systems engineering (see Section 1.1), we believe that the four-variable model is more suitable in this regard.

1.3.3 Existence of *SOF*

Methods for assessing the existence of *SOF* in the four-variable model have not received much attention in the literature. Of the few examples, (Lawford et al., 2000) give, without proof, a necessary condition for the existence of *SOF* in a functional variant of the four-variable model. In support of their claim, an example is presented where a pressure sensor is read by software and a reactor shutdown is initiated whenever the pressure value rises above a setpoint. It is shown with the interactive proof assistant PVS that in the particular conditions of the example no discrete implementation can meet the requirement. The result suggests that functional equality between requirements and implementations is too restrictive and tolerances must be allowed on the requirements, thus a case for using relations to model tolerances is made.

In the context of real-time systems, (Hu, 2008; Hu et al., 2009) address in a functional four-variable model the ability of software 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. Although in our approach we do not consider time explicitly, timing details can be dealt with by considering the members of the sets **M**, **C**, **I**, and **O** as functions of time (Parnas and Madey, 1995; Lawford et al., 2000; Peters, 2000).

In contrast with the works mentioned above, we address the need for existence conditions for *SOF* in the general, relational case of the four-variable model. The relational setting is more realistic as it can model the nondeterministic behaviours induced by hardware inaccuracies and tolerances on requirements. The results presented in the functional setting of these works can be obtained as particular cases of our relational results.

1.4 Thesis Approach and Contributions

To answer the question of implementability of system requirements in the four-variable model, we propose a mathematical basis for checking if an acceptable software specification *SOF* exists given the constraints imposed by the physical environment *NAT* and hardware interfaces *IN* and *OUT*. Our formalization of what it means for a software specification to be "acceptable" is based on the demonic calculus of relations, described in (Frappier, 1995), (Frappier et al., 1996), (Desharnais et al., 1997), and (Kahl, 2003b).

In Chapter 2 we introduce the mathematical concepts needed in the subsequent chapters. In particular, we introduce in the context of an algebra of concrete relations the demonic relational operators needed in the thesis.

In Chapter 3 we present the first theoretical contributions of the thesis. We use a diagram isomorphic to the four-variable model diagram to reduce notational verbosity. In this diagram, the relation that corresponds to *REQ* and *NAT* is factored through the relations that correspond to *IN*, *SOF*, and *OUT*. Since we adopt a demonic approach, the commutativity condition of the diagram is demonic refinement and composition of relations is demonic composition. Demonic refinement is thus our satisfaction relation between implementations and specifications. We prove a necessary and sufficient existence condition for the relation that corresponds to *SOF*, which we call a demonic mid factor, such that the diagram commutes. The existence condition for a demonic mid factor is a new result in relation algebra. Along the way, we also prove necessary and sufficient conditions for the existence of demonic left and, respectively, right factors. The results of this chapter will be applied to the four-variable model in the subsequent chapters, in order to answer the ques-

tion of implementability of system requirements. The demonic factorization results presented in this chapter are broader than safety-critical embedded systems; they are applicable wherever a diagram similar to the four-variable model diagram makes sense.

From an engineering perspective, a necessary condition for a specification REQ of the system requirements to be implementable is for REQ to specify only physically meaningful behaviours. To this end, (Parnas and Madey, 1995) defined the concept of feasibility of REQ with respect to the physical environment specified by NAT. Their formalization requires REQ to agree with NAT on at least one output for every input of NAT. This is problematic, however, as system designers may try to implement parts of REQ whose outputs do not obey the physical laws described by NAT. Clearly, such requirements specifications REQ are not fully implementable. To address this issue, in Chapter 4 we strengthen in the demonic calculus of relations the notion of feasibility of system requirements proposed by Parnas and Madey such that REQ specifies for every input possible in the environment only behaviours allowed by the environment.

Also in Chapter 4, we uncover a problem with the acceptability condition proposed by Parnas and Madey. In their formalization, a software specification SOF is acceptable although mismatches between the relations IN, SOF, and OUT are allowed that result in nondeterministic system behaviours which, for some inputs, sometimes produce expected outputs and some other times do not produce any results at all. Therefore, the acceptability condition proposed by Parnas and Madey may be seen as angelic, thus ensuring only partial correctness. To address these shortcomings, we give a new acceptability condition for system design and software specifications using the demonic calculus of relations, whose total correctness guarantees are more appropriate for safety-critical systems.

Assuming that the system requirements are feasible with respect to the physical environment, the existence of an acceptable software specification is conditioned only by the choice of input and output hardware devices. To this end, in Chapter 4 we also give a necessary and sufficient existence condition for an acceptable SOF. This condition has a constructive flavour and yields the

weakest (i.e., least restrictive, or least refined) software specification, which we regard as the software requirements.

A practical implication of the necessary and sufficient condition of Chapter 4 is that the input and output hardware interfaces are, in general, mutually dependent and changes to one may require changes to the other in order for an acceptable *SOF* to exist. In Chapter 5 we investigate a result analogous to the separation principle from control systems, which allows one to decompose the design of an optimal feedback control system into two independent tasks, an observer and a controller (Kalman, 1960). Similarly to this principle, we define the notions of observability (controllability) of the system requirements with respect to the input (output) interface and show that for a system that can be modelled by a functional four-variable model, observability and controllability allow for the separation of the design of the input and output interfaces. We also show that in the general, relational four-variable model we can obtain a similar effect by strengthening either observability or controllability. The two resulting implementability conditions are stronger than the necessary and sufficient condition of Chapter 4 and restrict the choice of input or output devices, but at the same time ensure their separability while still guaranteeing the existence of an acceptable *SOF*.

For the cases when a specification *REQ* of the system requirements is feasible with respect to a physical environment (i.e., only physically meaningful behaviours are specified), but an acceptable *SOF* does not exist, a typical engineering approach is to relax *REQ* by allowing tolerances. We show in Chapter 6 how the necessary and sufficient implementability condition of Chapter 4 can be used in the derivation of tolerances on the requirements for the shutdown system of a nuclear reactor such that the requirements become implementable.

The mathematical results of the thesis as well as the implementability analysis and tolerances for the example presented in Chapter 6 have been formalized and verified in the interactive proof assistant Coq. To not restrict the potential audience of the dissertation only to readers familiar with interactive theorem proving, in the body of the dissertation we adopt a traditional presentation style giving informal, but rigorous proofs. The formalization in

Coq is briefly presented in Appendix B, and a literate programming version with detailed explanations is available electronically at `www.cas.mcmaster.ca/~patcaslm/thesis/coq`.

Note: Parts of the dissertation have been published. The demonic factorization results of Chapter 3, as well as the demonic software acceptability definition and the necessary and sufficient existence condition for acceptable software of Chapter 4 have been presented in (Patcas et al., 2014b). An extended version of (Patcas et al., 2014b) which also includes parts from Chapters 2 and 6 is the object of (Patcas et al., 2014a), currently under review. The separability conditions for the input and output hardware interfaces given in Chapter 5 have been published in (Patcas et al., 2014c).

Chapter 2

Mathematical Preliminaries

In this chapter we introduce the mathematical concepts needed in the thesis. As seen in Section 1.2.4, there is a practical need for a relational four-variable model, thus the formal framework we adopt is that of relation algebra (Brink et al., 1997). Relation algebras usually treat relations at an abstract level. In this thesis we prefer concrete relations, that is, subsets of cartesian products, since the five relations in the four-variable model describe behaviours as input-output pairs. Moreover, being able to refer to elements in the domain or range of a relation gives a better engineering insight into the meaning of, and constraints on, the relations in the four-variable model. Although not impossible, this can become quite awkward in a language of abstract relations. However, since we would like to benefit from the large body of knowledge that is available in the realm of abstract relation algebras, we treat concrete relations as a particular case of abstract relations.

Another distinction that needs to be made is between homogeneous and heterogeneous relations. Homogeneous relations have their domains and ranges defined on the same universe; in contrast, heterogeneous relations are defined between two distinct universes, hence have a direction associated to them. For most applications related to software, the heterogeneous approach is preferable because its two-sorted language offers benefits similar to those of static typing in programming languages (Kahl, 2003b).

In particular, we will use the demonic calculus of relations (Frappier,

1995; Desharnais et al., 1997; Kahl, 2003b). For the reasons mentioned above, we will introduce the demonic calculus in the context of an algebra of concrete heterogeneous relations. To define this algebra, we first need to introduce posets, lattices and Boolean algebras.

2.1 Posets, Lattices, and Boolean Algebras

Let A be a nonempty set and $\sqsubseteq$ a binary relation on A. We say that $\sqsubseteq$ is respectively *reflexive*, *transitive*, *symmetric*, and *antisymmetric* if

$$x \sqsubseteq x\,, \qquad (\sqsubseteq \text{ reflexive})$$
$$x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z\,, \qquad (\sqsubseteq \text{ transitive})$$
$$x \sqsubseteq y \Rightarrow y \sqsubseteq x\,, \qquad (\sqsubseteq \text{ symmetric})$$
$$x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y\,, \qquad (\sqsubseteq \text{ antisymmetric})$$

for any $x, y, z \in A$.

A relation $\sqsubseteq$ that is reflexive and transitive is called a *preorder*. If $\sqsubseteq$ is also symmetric, then it is called an *equivalence relation*. If $\sqsubseteq$ is an antisymmetric preorder, then $\sqsubseteq$ is a *partial order*.

Definition 2.1. A partially ordered set, or *poset*, is a structure $(A, \sqsubseteq)$ where A is a nonempty set and $\sqsubseteq$ is a partial order on A.

Let X be a set. The *least element* of X is an element $x \in X$ such that $x \sqsubseteq y$ for any $y \in X$. Similarly, $x \in X$ is the *greatest element* of X if $y \sqsubseteq x$ for any $y \in X$. If they exist, the least and greatest elements of a set are unique.

The least element of the poset $(A, \sqsubseteq)$ is called the *bottom element* and is denoted by $\mathbb{\bot}$. The greatest element of A is called the *top element* and is denoted by $\mathbb{\top}$. A poset is *bounded* if it has both a top and a bottom, that is, $\mathbb{\bot} \sqsubseteq a \sqsubseteq \mathbb{\top}$ for any $a \in A$.

In a poset $(A, \sqsubseteq)$, an element $a \in A$ is a *lower bound* of a subset X of A if $a \sqsubseteq x$ for any $x \in X$. The *greatest lower bound* (or *infimum*, or *meet*) of X, denoted $\bigsqcap X$, is the greatest element of the set of lower bounds of X. Dually, an element $a \in A$ is an *upper bound* of a subset X of A if $x \sqsubseteq a$ for any

$x \in X$. The *least upper bound* (or *supremum*, or *join*) of X, denoted $\bigsqcup X$, is the least element of the set of upper bounds of X. If they exist, the greatest lower bound and least upper bound of a subset X of A are unique.

Definition 2.2. A poset $(A, \sqsubseteq)$ is called a *lattice* if and only if every subset $\{x, y\}$ of A has a join $x \sqcup y = \bigsqcup\{x, y\}$ and a meet $x \sqcap y = \bigsqcap\{x, y\}$.

A direct consequence of this definition is that a poset $(A, \sqsubseteq)$ is a lattice if and only if $\bigsqcup X$ and $\bigsqcap X$ exist for every finite nonempty subset X of A.

Since the meet and join in a lattice $(A, \sqsubseteq)$ are defined for any pair of elements of A, they can be seen as binary operations on A and thus form an algebra on A. The join and meet operations on a lattice $(A, \sqsubseteq)$ then have the following properties:

$$
\begin{aligned}
&(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z) \text{ and } (x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)\,, && \text{(associativity)}\\
&x \sqcup y = y \sqcup x \text{ and } x \sqcap y = y \sqcap x\,, && \text{(commutativity)}\\
&x \sqcup x = x \text{ and } x \sqcap x = x\,, && \text{(idempotence)}\\
&(x \sqcup y) \sqcap x = x \text{ and } (x \sqcap y) \sqcup x = x\,, && \text{(absorption)}
\end{aligned}
$$

for any $x, y, z \in A$.

The connection between the algebraic meet and join operations on a lattice $(A, \sqsubseteq)$ and the partial order $\sqsubseteq$ is given by the following relationships:

$$
\begin{aligned}
x \sqcup y = y &\iff x \sqsubseteq y\\
x \sqcap y = x &\iff x \sqsubseteq y
\end{aligned}
$$

for any $x, y \in A$. Hence a lattice can be seen a poset $(A, \sqsubseteq)$ as well as an algebraic structure $(A, \sqcup, \sqcap)$.

A lattice $(A, \sqsubseteq)$ is said to be:

- *bounded* if it has a bottom element $\perp\!\!\!\perp$ (i.e., $\perp\!\!\!\perp = \bigsqcup \varnothing = \bigsqcap A$) and a top element $\top\!\!\!\top$ (i.e., $\top\!\!\!\top = \bigsqcup A = \bigsqcap \varnothing$);
- *complete* if $\bigsqcup X$ and $\bigsqcap X$ exist in A for every $X \subseteq A$. In particular, the join and meet must exist for an empty X as well as for an infinite

X. Every complete lattice has a bottom $\perp\!\!\!\perp$ and a top $\top\!\!\!\top$ that satisfy the following properties:

$$\perp\!\!\!\perp = \bigsqcap A \text{ and } \perp\!\!\!\perp = \bigsqcup \varnothing\,,$$

$$\top\!\!\!\top = \bigsqcap A \text{ and } \top\!\!\!\top = \bigsqcap \varnothing\,;$$

- *complemented* if it is bounded and for every element $x \in A$ there exists an element $\overline{x}$, called the *complement* of x, such that

$$x \sqcup \overline{x} = \top\!\!\!\top \text{ and } x \sqcap \overline{x} = \perp\!\!\!\perp\,;$$

- *distributive* if for every $x, y, z \in A$ the following two properties hold:

$$x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)\,, \qquad (\sqcup \text{ distributivity})$$
$$x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)\,. \qquad (\sqcap \text{ distributivity})$$

  In a distributive lattice, the complement of an element is unique, if it exists.

Definition 2.3. A poset $(A, \sqsubseteq)$ is called a *meet semilattice* if $x \sqcap y$ exists in A for any x and y in A. A *join semilattice* is defined similarly.

Definition 2.4. A *Boolean algebra*, or a *Boolean lattice*, is a complemented distributive lattice.

In a Boolean algebra, the complement is an involution:

$$\overline{\overline{x}} = x\,.$$

Boolean algebras satisfy the following laws:

$$\overline{x \sqcup y} = \overline{x} \sqcap \overline{y} \text{ and } \overline{x \sqcap y} = \overline{x} \sqcup \overline{y}\,. \qquad (\text{de Morgan})$$

An important concept that we will use to compare our theoretical results with results from relation algebra literature is that of a dual lattice.

Definition 2.5. The *dual* of a lattice $(A, \sqsubseteq)$ is the lattice $(A, \sqsupseteq)$.

The dual lattice $(A, \sqsupseteq)$ is complete, distributive, or Boolean if and only if the lattice $(A, \sqsubseteq)$ is complete, distributive, or Boolean. In general, the following *duality principle* holds (Back and von Wright, 1998): if ϕ is a statement about a lattice $(A, \sqsubseteq)$, then the dual statement about the dual lattice $(A, \sqsupseteq)$ is obtained from ϕ by interchanging $\sqsubseteq$ and $\sqsupseteq$, $\sqcup$ and $\sqcap$, $\top\!\!\top$ and $\bot\!\!\bot$, while leaving the complement unchanged.

2.2 Abstract Heterogeneous Relation Algebra

In defining an algebra of abstract heterogeneous relations, we use the definition proposed by (Kahl, 2003b). This definition is a variation of the definition based on category theory initially proposed in (Schmidt et al., 1997).

Definition 2.6. An *abstract heterogeneous relation algebra* is formed of *objects* $\mathcal{A}, \mathcal{B}, \ldots$ and *relations* $P, Q, R, \ldots$, with the following operations and properties:

- Every relation R has a *source* object, denoted $\mathsf{source}(R)$, and a *target* object, denoted $\mathsf{target}(R)$. For any two objects $\mathcal{A}$ and $\mathcal{B}$, $\mathcal{A} \leftrightarrow \mathcal{B}$ denotes the set of all relations with $\mathsf{source}(R) = \mathcal{A}$ and $\mathsf{target}(R) = \mathcal{B}$. The notation $R : \mathcal{A} \leftrightarrow \mathcal{B}$ denotes a relation R that has $\mathcal{A}$ as source and $\mathcal{B}$ as target;

- The set $\mathcal{A} \leftrightarrow \mathcal{B}$ between two objects $\mathcal{A}$ and $\mathcal{B}$ forms a Boolean algebra $(\mathcal{A} \leftrightarrow \mathcal{B}, \sqcup, \sqcap, \overline{}, \bot\!\!\bot_{\mathcal{A},\mathcal{B}}, \top\!\!\top_{\mathcal{A},\mathcal{B}})$, where the join $\sqcup$ is called *union*, the meet $\sqcap$ is called *intersection*, every relation $R : \mathcal{A} \leftrightarrow \mathcal{B}$ has a *complement* $\overline{R} : \mathcal{A} \leftrightarrow \mathcal{B}$, the bottom element $\bot\!\!\bot_{\mathcal{A},\mathcal{B}}$ is the *empty* relation between $\mathcal{A}$ and $\mathcal{B}$, and the top element $\top\!\!\top_{\mathcal{A},\mathcal{B}}$ is the *universal* relation between $\mathcal{A}$ and $\mathcal{B}$. The indices for the empty and universal relations can be omitted if they can be inferred from the context. The partial order $\sqsubseteq$ on $\mathcal{A} \leftrightarrow \mathcal{B}$ is called *inclusion*. The operations $\sqcup$ and $\sqcap$, and partial order $\sqsubseteq$ between relations P and Q are defined if and only if $\mathsf{source}(P) = \mathsf{source}(Q)$ and $\mathsf{target}(P) = \mathsf{target}(Q)$;

- The *composition* of two relations P and Q is the operation $P \fatsemi Q$ defined if and only if $\mathsf{target}(P) = \mathsf{source}(Q)$ and is an element of $\mathsf{source}(P) \leftrightarrow \mathsf{target}(Q)$;
- For every object $\mathcal{A}$ there is an *identity* element $\mathbb{I}_{\mathcal{A}}$ such that for every relation R it is the case that $\mathbb{I}_{\mathsf{source}(R)} \fatsemi R = R$ and $R \fatsemi \mathbb{I}_{\mathsf{target}(R)} = R$;
- The *converse* of a relation $P : \mathcal{A} \leftrightarrow \mathcal{B}$ is the relation $P^{\smile} : \mathcal{B} \leftrightarrow \mathcal{A}$. For all objects $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, and relations $P : \mathcal{A} \leftrightarrow \mathcal{B}$, $Q : \mathcal{B} \leftrightarrow \mathcal{C}$ and $R : \mathcal{A} \leftrightarrow \mathcal{C}$, the *Dedekind rule* holds:

$$R \sqcap P \fatsemi Q \sqsubseteq \left(P \sqcap R \fatsemi Q^{\smile}\right) \fatsemi \left(Q \sqcap P^{\smile} \fatsemi R\right) .$$

Many properties of relations can be proven from the above definition. In the thesis we will use some of the properties given in (Schmidt et al., 1997):

- composition is associative:

$$P \fatsemi (Q \fatsemi R) = (P \fatsemi Q) \fatsemi R ;$$

- composition distributes over intersection:

$$Q \fatsemi (R \sqcap P) = Q \fatsemi R \sqcap Q \fatsemi P ;$$

- composition distributes over union:

$$Q \fatsemi (R \sqcup P) = Q \fatsemi R \sqcup Q \fatsemi P ;$$

- conversion distributes over intersection:

$$(P \sqcap Q)^{\smile} = P^{\smile} \sqcap Q^{\smile} ;$$

- conversion distributes over union:

$$(P \sqcup Q)^{\smile} = P^{\smile} \sqcup Q^{\smile} ;$$

- conversion antidistributes over composition:

$$(P \fatsemi Q)^{\smile} = Q^{\smile} \fatsemi P^{\smile};$$

- conversion is an involution:

$$(R^{\smile})^{\smile} = R;$$

- the complement and converse commute:

$$\overline{R^{\smile}} = \left(\overline{R}\right)^{\smile}.$$

2.3 Concrete Heterogeneous Relation Algebra

Concrete relation algebras are standard models of abstract relation algebras. In a concrete relation algebra, objects are *sets* and relations are subsets of cartesian products between sets. Such relations are called *concrete relations* and are the same as the usual relations used in discrete mathematics. As such, an abstract relation $R : \mathcal{A} \leftrightarrow \mathcal{B}$ becomes the concrete relation $R \subseteq \mathcal{A} \times \mathcal{B}$, where $\mathcal{A}$ and $\mathcal{B}$ are sets.

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 $R = \{(a, b) \in A \times B \mid R_{pred}(a, b)\}$, 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: $R(a) = \{b \in B \mid (a, b) \in R\}$;

- *preimage set* of $b \in B$ under R: $R^{\smile}(b) = \{a \in A \mid (a, b) \in R\}$.

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 Section 1.2.4, 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 modelling nondeterminism: the image set of an element in the domain of a non-univalent relation denotes all the possible results for that input. Functional relations model deterministic behaviours since the image sets of the elements in their domains are all singletons. Because we treat functions as a particular case of relations, determinism becomes a special case of nondeterminism.

In addition to the nondeterminism caused by input/output hardware inaccuracies and tolerances on requirements, there is another form of nondeterminism caused by the composition of partial specifications. As discussed in Section 1.3.1, the main approaches to deal with such nondeterministic behaviours are angelic and demonic. In the angelic approach, specifications (implementations) that allow "bad" behaviours for some inputs are permitted as long as they also allow "good" behaviours for those inputs. In contrast, in the demonic approach, specifications (implementations) that allow "bad" behaviours are not permitted at all. When developing safety-critical systems it is always wise to plan for the worst, therefore we argue that the demonic approach is more suitable and use the demonic calculus of relations (Frappier, 1995; Desharnais et al., 1997; Kahl, 2003b). Because the operations in the demonic calculus are usually defined in terms of their angelic counterparts, we first present the angelic operations and then the demonic ones.

2.3.1 Angelic Calculus

The angelic operations are the usual relational operations.

Intersection and Union

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

$$P \cup Q = \{(a, b) \in A \times B \mid (a, b) \in P \vee (a, b) \in Q\}\,.$$

Inclusion

Definition 2.7. A relation $P \subseteq A \times B$ is *contained*, or *included*, in a relation $Q \subseteq A \times B$, written $P \subseteq Q$, if and only if for every $(a, b) \in P$ it is also the case that $(a, b) \in Q$.

Relational inclusion $\subseteq$ 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 universal relation between A and B, $\top_{A,B} = \{(a, b) \in A \times B \mid \mathsf{true}\}$, and the bottom element is the empty relation between A and B, $\bot_{A,B} = \{(a, b) \in A \times B \mid \mathsf{false}\}$.

Relational inclusion is used as a *refinement* ordering between specifications and/or implementations in, for example, (Hoare and He, 1986; Hoare and He, 1987; Hoare et al., 1987). Relational inclusion is also known as *partial correctness* in (Kahl, 2003b), where an elegant mathematical explanation is given as to why the satisfaction and refinement relations between implementations and specifcations are equivalent concepts when relations are used for describing both specifications and implementations. Therefore, 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 for any specification means that implementations that do not produce any results are always acceptable. This is also not something desirable for a safety-critical system.

Composition

Definition 2.8. The *composition* of two relations $P \subseteq A \times B$ and $Q \subseteq B \times C$ is the relation:

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

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

Residuals

Relational composition and inclusion induce two residuation operations, the left and right residuals (Hoare and He, 1985; Hoare and He, 1986; Hoare and He, 1987; Hoare et al., 1987; Schmidt and Ströhlein, 1993; Frappier, 1995; Brink et al., 1997; Kahl, 2003b).

Definition 2.9. Given 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 $Y \mathbin{;} Q \subseteq R$, where $Y \subseteq A \times B$ is the unknown:

$$Y \mathbin{;} Q \subseteq R \Leftrightarrow Y \subseteq R/Q \,.$$

The value of R/Q is:

$$\begin{aligned} R/Q = \overline{\overline{R} \mathbin{;} Q^{\smile}} &= \{(a,b) \in A \times B \mid \forall c \in C \mathbin{\bullet} (b,c) \in Q \Rightarrow (a,c) \in R\} \\ &= \{(a,b) \in A \times B \mid Q(b) \subseteq R(a)\} \,. \end{aligned} \tag{2.1}$$

Definition 2.10. 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 $P \mathbin{;} X \subseteq R$, where $X \subseteq B \times C$ is the unknown:

$$P \mathbin{;} X \subseteq R \Leftrightarrow X \subseteq P\backslash R \,.$$

The value of $P\backslash R$ is:

$$\begin{aligned} P\backslash R = \overline{P^{\smile} \mathbin{;} \overline{R}} &= \{(b,c) \in B \times C \mid \forall a \in A \mathbin{\bullet} (a,b) \in P \Rightarrow (a,c) \in R\} \\ &= \left\{(b,c) \in B \times C \;\middle|\; P^{\smile}(b) \subseteq R^{\smile}(c)\right\} \,. \end{aligned} \tag{2.2}$$

The precedence of / and \ is the same as the precedence of 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. The left and right residuals are different because relational composition is not commutative.

C. A. R. Hoare and his group at Oxford were among the first to advocate the importance of the relational residuals to software development[1] (Hoare and

[1]Hoare and He use a different notation than ours; we follow the conventions standardized in RelMiCS (Brink et al., 1997).

He, 1985; Hoare and He, 1986; Hoare and He, 1987; Hoare et al., 1987). The residuals are useful when a specification is refined by a composition of two specifications of which one is unknown. 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.

Examples of relational residuals will be given in the next section when they are compared with their demonic counterparts.

2.3.2 Demonic Calculus

We now present the demonic relational operations that we will use in the subsequent chapters of the thesis and motivate their suitability for safety-critical systems compared to their angelic counterparts. We introduce the demonic operators similarly to (Frappier, 1995),(Desharnais et al., 1997), and (Kahl, 2003b), however, instead of the abstract algebraic style we favour concrete relations, which we believe are more suited for an engineering audience. The demonic operators have the same precedence as their angelic counterparts.

Domain and Range Restrictions

Before presenting the demonic operations, we introduce notational abbreviations 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 concrete relations.

Definition 2.11. The domain restriction of a relation $P \subseteq A \times B$ to a set $A' \subseteq A$ is the relation $P\big|_{A'} = \{(a, b) \in P \mid a \in A'\}$.

Definition 2.12. 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 (Mili et al., 1987).

We will use particular cases of domain and range restrictions, where the domain or range of a relation is restricted to the domain or range of another

relation. Before introducing these restrictions, we need to introduce relational *vectors* (Schmidt et al., 1997; Schmidt, 2011), which will allow us to make the connection between the abstract and concrete formulations of these particular domain and range restrictions.

Definition 2.13. A relation v satisfying $v = v \,;\, \mathbb{T}$ is called a vector.

Vectors are useful for describing the domain and range of a relation in a purely relation-algebraic way. Clearly, the domain of a vector $v \,;\, \mathbb{T}$ is equal to the domain of v, while the range of $v \,;\, \mathbb{T}$ is equal to the target of v. Thus, the vector $v \,;\, \mathbb{T}$ "describes" the domain of v. Dually, the range of v is described by the vector $v^\smile \,;\, \mathbb{T}$. The same effect can be obtained by using the converse vectors. The converse of $v \,;\, \mathbb{T}$ is $\mathbb{T} \,;\, v^\smile$ and describes the domain of v. The converse of $v^\smile \,;\, \mathbb{T}$ is $\mathbb{T} \,;\, v$, which describes the range of v.

The domain and range restrictions we will use are:

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

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

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

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

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

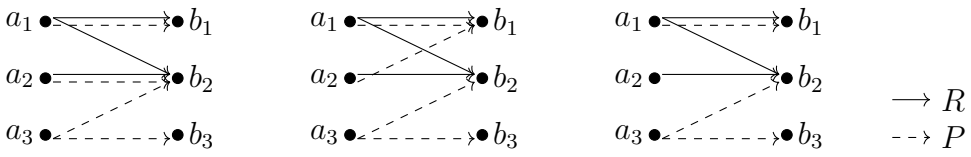
$$P\big|^{\mathsf{dom}(R)} = P \cap \mathbb{T}_{A,C} \,;\, R^\smile = \{(a,b) \in P \mid b \in \mathsf{dom}\,(R)\};$$

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

$$P\big|^{\mathsf{ran}(R)} = P \cap \mathbb{T}_{A,C} \,;\, R = \{(a,b) \in P \mid b \in \mathsf{ran}\,(R)\}.$$

Demonic refinement

Definition 2.14. 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 $R ; \mathbb{T}_{B,B} \subseteq P ; \mathbb{T}_{B,B}$ and $R ; \mathbb{T}_{B,B} \cap P \subseteq$



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

Figure 2.1: Examples of demonic refinement

R. Using the domain vectors and domain restrictions, these conditions become:

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

(ii) and, respectively, $P\big|_{\mathsf{dom}(R)} \subseteq R$.

Consider the relations P and R in Figure 2.1: in Figure 2.1a, P refines R; in Figure 2.1b, P does not refine R because $(a_2, b_1) \in P$ but $(a_2, b_1) \notin R$; and, in Figure 2.1c, P does not refine R because $\mathsf{dom}\,(R) \not\subseteq \mathsf{dom}\,(P)$.

(Maddux, 1996) made the connection between the approaches to nondeterminism and refinement in variations of Dijkstra's weakest precondition calculus such as those in (Back, 1981), (Morgan and Robinson, 1987), (Morris, 1987), or (Back and von Wright, 1992), and those in relation-algebraic formalisms such as (Mili, 1983; Mili et al., 1987), (Boudriga et al., 1992), and (Desharnais et al., 1995; Frappier, 1995; Frappier et al., 1996; Desharnais et al., 1997; Kahl, 2003b). This connection reveals that demonic refinement and the refinement orderings in the works just mentioned are the same. As such, demonic refinement appears under various guises in the literature:

- "more defined than" in (Mili et al., 1987; Boudriga et al., 1992);
- total correctness in (Mili, 1983; Desharnais et al., 1997; Kahl, 2003b);
- demonic refinement in (Desharnais et al., 1995; Frappier, 1995; Frappier et al., 1996; Desharnais et al., 1997; Kahl, 2003b).

Demonic refinement is a partial order and induces a complete join semilattice, usually referred to as the *demonic lattice* (Boudriga et al., 1992; Desharnais et al., 1995; Frappier, 1995; Frappier et al., 1996; Desharnais et al.,

1997; Kahl, 2003b)[2]. 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 total relations which specify termination everywhere; the minima of this set are the ideal implementations (i.e., total functions). Demonic union $\sqcup$ is the join operation of the demonic lattice and will not be used in this thesis. The meet operation is the demonic intersection $\sqcap$, which is not always defined because the demonic lattice is a join semilattice.

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 produces incorrect results or no results at all).

Compared to the angelic refinement (i.e., relational inclusion $\subseteq$), demonic refinement does not allow empty implementations for non-empty specifications. 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 behaviour outside the domain of a specification is the best thing to do. When we give a demonic semantics to the four-variable model in Chapter 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.

[2]The reader should note that the demonic refinement ordering used in (Boudriga et al., 1992) and (Frappier, 1995) is the converse of the usual demonic refinement ordering, thus the dual of the demonic lattice is used in these works.

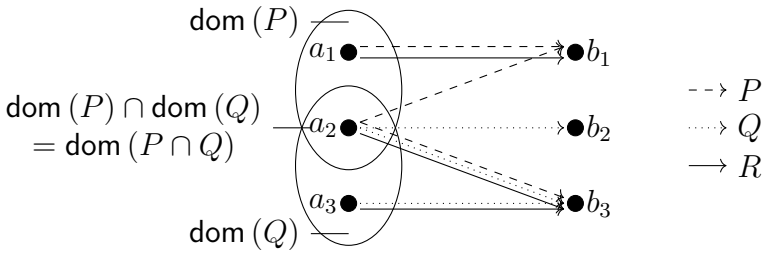


Figure 2.2: Example of demonic intersection

Demonic intersection

Definition 2.15. Two relations $P \subseteq A \times B$ and $Q \subseteq A \times B$ are compatible if and only if

$$\mathsf{dom}\,(P) \cap \mathsf{dom}\,(Q) \subseteq \mathsf{dom}\,(P \cap Q)\ . \tag{2.3}$$

The meaning of (2.3) is that for every input in common, P and Q should have at least one output in common.

Definition 2.16. The demonic intersection (demonic meet) of P and Q, denoted as $P \sqcap Q$, is defined if and only if P and Q are compatible. If the demonic intersection of P and Q is defined, then its value is:

$$\begin{aligned} P \sqcap Q &\succeq (P \cap Q) \cup (P \cap \overline{Q \,;\, \top}) \cup (\overline{P \,;\, \top} \cap Q) \\ &\succeq (P \cap Q) \cup P\big|_{\overline{\mathsf{dom}(Q)}} \cup Q\big|_{\overline{\mathsf{dom}(R)}}\,. \end{aligned} \tag{2.4}$$

The symbol $\succeq$, called the "venturi tube" (Kahl, 2003b), 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 (2.4) is that $P \sqcap Q$ captures the behaviour 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, $P \sqcap Q = \{(a_2, b_3), (a_1, b_1), (a_3, b_3)\}$ (Figure 2.2).

(a) Angelic composition (b) Demonic composition

Figure 2.3: Demonic vs. angelic composition

Demonic composition

Definition 2.17. The demonic composition of two relations $P \subseteq A \times B$ and $Q \subseteq B \times C$ is the relation

$$P \mathbin{▫} Q = P \mathbin{;} Q \cap \overline{P \mathbin{;} \overline{Q \mathbin{;} \top_{C,C}}} = \{(a, c) \in P \mathbin{;} Q \mid P(a) \subseteq \mathsf{dom}\,(Q)\}\,.$$

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 explains, perhaps the best, 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 Figure 2.3. Here, $P \mathbin{;} Q$ allows the dead end (a_1, b_2) because there is a chance that a_1 will reach c_1 via b_1. On the other hand, $P \mathbin{▫} Q$ is empty because there is the possibility that an implementation will get stuck at b_2 and will not reach c_1. In practice, this means that an implementation of $P \mathbin{;} Q$ will sometimes produce a result and sometimes it will not.

Demonic composition is a fully associative operation (Backhouse and van der Woude, 1993): any three relations P, Q, R satisfy $P \mathbin{▫} (Q \mathbin{▫} R) = (P \mathbin{▫} Q) \mathbin{▫} R$.

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.

Definition 2.18. 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 the largest solution with respect to $\sqsubseteq$ of the inequation $Y \mathbin{\square} Q \sqsubseteq R$, where $Y \subseteq A \times B$ is the unknown:

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

A solution Y, 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. In fact, a consequence of Definition 2.18 is that $R \mathbin{/\!\!/} Q$ is defined if and only if a demonic left factor Y exists.

Several necessary and sufficient conditions for the definedness of $R \mathbin{/\!\!/} Q$ can be found in the literature, such as, if converted to our notation:

$$\mathsf{dom}\,(R) \subseteq \mathsf{dom}\left((R/Q)\,\big|^{\mathsf{dom}(Q)}\right) \tag{2.5}$$

in (Desharnais et al., 1995) and (Frappier, 1995); or,

$$\mathsf{dom}\,(R) \subseteq \mathsf{dom}\,((R/Q) \mathbin{;} Q) \tag{2.6}$$

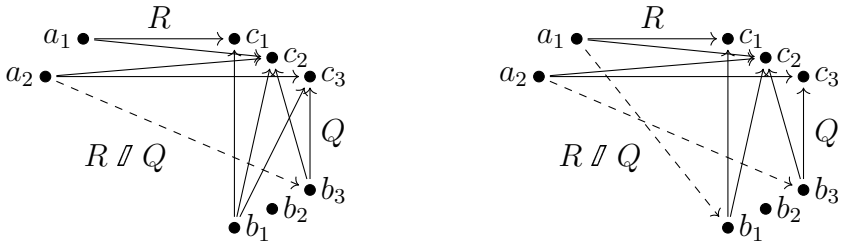
in (Desharnais et al., 1997) and (Kahl, 2003b). In Section 3.1 we will prove a new necessary and sufficient condition for the existence of a demonic left factor expressed in predicate logic that offers better engineering insight than the aforementioned conditions.

If the demonic left residual $R \mathbin{/\!\!/} Q$ is defined, then its value is obtained by restricting the range of the angelic left residual R/Q to the domain of Q, that is, $R \mathbin{/\!\!/} Q \simeq R/Q \cap \top \mathbin{;} Q^{\smile}$ (Desharnais et al., 1993; Frappier, 1995; Desharnais et al., 1995; Desharnais et al., 1997; Kahl, 2003b). An equivalent formulation in our notation is:

$$\begin{aligned} R \mathbin{/\!\!/} Q &\simeq (R/Q)\,\big|^{\mathsf{dom}(Q)} \\ &\simeq \{(a,b) \in A \times B \mid b \in \mathsf{dom}\,(Q) \wedge Q(b) \subseteq R(a)\} \,. \end{aligned} \tag{2.7}$$

The demonic residuals are partial operations whose results are not always

[3]The demonic left residual is called the conjugate kernel in (Desharnais et al., 1993).



(a) $R /\!/\!/ Q$ is not well defined (b) $R /\!/\!/ Q$ is well defined

Figure 2.4: Definedness of the demonic left residual

well defined. Non-definedness of the demonic residuals does not necessarily mean emptiness. This important point is made explicit in Figure 2.4a. Here, the demonic left residual of a relation R by a relation Q calculated using (2.7) is
$R /\!/\!/ Q = \{(a_2, b_3)\}$. Although not empty, this demonic left residual is not well defined because it does not satisfy Definition 2.18. The reason for this is that $(R /\!/\!/ Q)\square Q = \{(a_2, c_2), (a_2, c_3)\}$ is not a demonic refinement of R since $\mathsf{dom}\,(R) \not\subseteq \mathsf{dom}\,((R /\!/\!/ Q)\square Q)$. Another way to check the definedness of $R /\!/\!/ Q$ would be to use either of the conditions (2.5) and (2.6), or the condition we will give in Section 3.1. Figure 2.4b illustrates an example where the demonic left residual is well defined.

Definition 2.19. The demonic right residual of a relation $R \subseteq A \times C$ by a relation $P \subseteq A \times B$, denoted $P \setminus\!\!\setminus R$, is the largest solution with respect to $\sqsubseteq$ of the inequation $P\square X \sqsubseteq R$, where $X \subseteq B \times C$ is the unknown:

$$P\square X \sqsubseteq R \Leftrightarrow X \sqsubseteq P \setminus\!\!\setminus R\,.$$

A solution X, called a *demonic right factor* of R through P, does not always exist. By Definition 2.19, the demonic right residual $P \setminus\!\!\setminus R$ is defined if and only if a demonic right factor X exists.

The definedness conditions for $P \setminus\!\!\setminus R$ given in (Frappier, 1995), (Desharnais et al., 1995) and (Kahl, 2003b) can all be converted to the following

common form in our notation (see Appendix A for details):

$$\mathsf{dom}\,(R) \subseteq \mathsf{dom}\,(P) \wedge \top_{B,C} \subseteq \left(P\big|_{\mathsf{dom}(R)} \backslash R\right) ; \top_{C,C}\,. \tag{2.8}$$

In the works mentioned above, this condition is stated only as sufficient. In Section 3.2 we will prove a necessary and sufficient condition formulated in predicate logic that offers a better engineering insight than (2.8). Our condition turns out to be equivalent to (2.8). Consequently, (2.8) is necessary and sufficient as well.

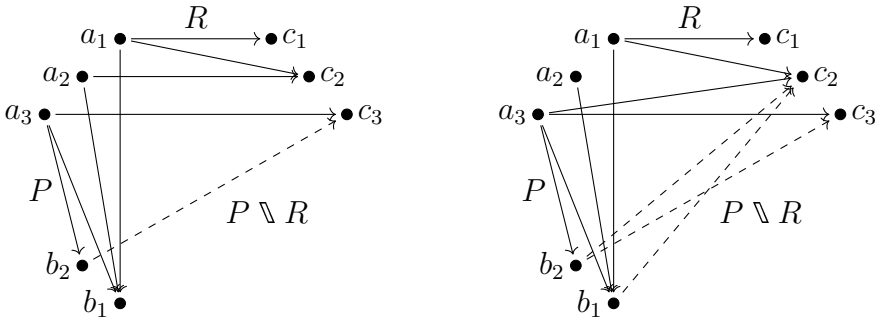
If the demonic right residual is defined, then its value is:

$$\begin{aligned} P \setminus\!\!\setminus R &\rightrightarrows \left(P\big|_{\mathsf{dom}(R)} \backslash R\right)\Big|_{\mathsf{ran}\left(P\big|_{\mathsf{dom}(R)}\right)} \\ &\rightrightarrows \left\{(b,c) \in B \times C \;\middle|\; b \in \mathsf{ran}\left(P\big|_{\mathsf{dom}(R)}\right) \wedge \left(P\big|_{\mathsf{dom}(R)}\right)^{\smile}(b) \subseteq R^{\smile}(c)\right\}. \end{aligned} \tag{2.9}$$

Several alternative definitions for the value of $P \setminus\!\!\setminus R$ are given in (Frappier, 1995),(Desharnais et al., 1995), and (Kahl, 2003b). In Appendix A we show that these definitions are equivalent to our definition (2.9).

Similarly to the demonic left residual, the demonic right residual does not have to be empty to be undefined. Figure 2.5a illustrates such a case. In this example, the value of the demonic right residual calculated using (2.9) is $P \setminus\!\!\setminus R = \{(b_2, c_3)\}$. Nevertheless, $P \setminus\!\!\setminus R$ is undefined because $P \mathbin{\square} (P \setminus\!\!\setminus R) = \perp\!\!\!\perp$ does not demonically refine R. Of course, (2.8) is also not satisfied in this case. In Figure 2.5b, the demonic right residual is well defined.

It is worth explaining why the demonic residuals are more suitable for safety-critical applications. Let us consider the relations depicted in Figure 2.6. In this example, 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



(a) $P \setminus\!\setminus R$ is not well defined (b) $P \setminus\!\setminus R$ is well defined

Figure 2.5: Definedness of the demonic right residual

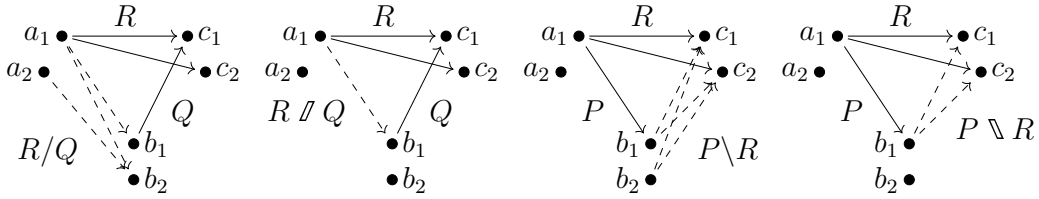


Figure 2.6: Demonic vs. angelic residuals

$P \setminus\!\setminus R$, asks its implementations to deal with b_2.

2.4 Covers and Equivalence Kernels

To answer the question of implementability, we need to a way to tell how much information about the monitored variables an implementation loses and also how accurate are the outputs produced by an implementation compared to what is required of the system. To this end, in this section we introduce the notion of a cover induced by a relation and that of the kernel of a function, along with some of their properties. In subsequent chapters, we will study implementability in both the general, relational four-variable model as well as in the particular case of a functional four-variable model. Covers are needed in the relational setting, while kernels are their counterparts in the functional setting.

A *cover* of a set A is a family $\mathcal{C} = \{C_\alpha \subseteq A \mid \alpha \in \mathcal{I}\}$ where α is an index in some index set $\mathcal{I}$, $A = \bigcup_{\alpha\in\mathcal{I}} C_\alpha$, and the subsets C_α of A, called the *cells*

of $\mathcal{C}$, are not necessarily pairwise disjoint. We denote the set of all covers of a set A by $\mathsf{Cov}\,(A)$.

Definition 2.20. A cover $\mathcal{C} \in \mathsf{Cov}\,(A)$ *refines* a cover $\mathcal{D} \in \mathsf{Cov}\,(A)$ if and only if every cell of $\mathcal{C}$ is contained in a cell of $\mathcal{D}$:

$$\mathcal{C} \leq \mathcal{D} \stackrel{\text{def}}{=} \forall A' \in \mathcal{C} \bullet \exists A'' \in \mathcal{D} \bullet A' \subseteq A'' \,.$$

If $\mathcal{C} \leq \mathcal{D}$, we say that $\mathcal{C}$ is "finer" than $\mathcal{D}$, or $\mathcal{D}$ is "coarser" than $\mathcal{C}$. Refinement of covers is a preorder (i.e., a reflexive and transitive ordering relation).

Any relation induces a cover on its domain, defined as follows.

Definition 2.21. (Wonham, 2013) A relation $R \subseteq A \times B$ induces a cover on $\mathsf{dom}\,(R)$ whose cells, indexed by $\mathsf{ran}\,(R)$, are the image sets of the elements in $\mathsf{ran}\,(R)$ under the converse of R:

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

A *partition* is a cover whose cells are pairwise disjoint. The set of all partitions of a set A is denoted by $\mathsf{Par}\,(A)$. Refinement of covers becomes a partial order in the particular case of partitions. Moreover, $\mathsf{Par}\,(A)$ is a complete lattice with refinement of covers as the partial order. The set of all equivalence relations on a set A, denoted by $\mathsf{Eq}\,(A)$, is a complete lattice with set inclusion as the partial order. The following bijective mapping $\theta : \mathsf{Par}\,(A) \to \mathsf{Eq}\,(A)$ can be defined between the partitions and equivalence relations on A:

$$\theta(\pi) = \{(a_1, a_2) \in A \times A \mid \exists A' \in \pi \bullet \{a_1, a_2\} \subseteq A'\} \,.$$

As such, the lattices $(\mathsf{Par}\,(A), \leq)$ and $(\mathsf{Eq}\,(A), \subseteq)$ are isomorphic and we can talk about partitions and equivalence relations interchangeably (Burris and Sankappanavar, 1981). In particular, an equivalence relation on a set partitions that set into *equivalence classes*. Also, refinement of covers can be used to order equivalence relations since partitions are a particular case of covers. Consequently, by Definition 2.20, an equivalence relation $\theta_1 \in \mathsf{Eq}\,(A)$ refines another equivalence relation $\theta_2 \in \mathsf{Eq}\,(A)$ if and only if every equivalence class

of θ_1 is contained in an equivalence class of θ_2.

A particular equivalence relation in $\mathsf{Eq}(A)$ is the equivalence relation induced by a function on its domain.

Definition 2.22. Let $f : A \to B$. The equivalence relation induced by f on its domain is called the *equivalence kernel* of f and is defined as follows:

$$\begin{aligned}\mathsf{ker}(f) &= f \fatsemi f^{\smile} \\ &= \{(a_1, a_2) \in A \times A \mid \exists b \in B \bullet (a_1, b) \in f \wedge (a_2, b) \in f\} \\ &= \{(a_1, a_2) \in A \times A \mid f(a_1) = f(a_2)\}\,.\end{aligned}$$

The equivalence kernel of a function f partitions the domain of f into equivalence classes (i.e., pairwise disjoint cells) such that every equivalence class contains all the elements in $\mathsf{dom}(f)$ that have the same image in $\mathsf{ran}(f)$. Consequently, kernels are similar to covers. In fact, in the functional case, the partition that corresponds to $\mathsf{ker}(f)$ is exactly the same as $\mathsf{cov}(f)$. In the relational case disjointness of the cells is lost.

Chapter 3

Demonic Factorization of Relations

In this chapter we present the first theoretical results of the thesis. In order to answer the question we posed about the implementability of system requirements, we are interested in existence conditions for the dotted arrows in the commutative diagram depicted in Figure 3.1, which is isomorphic to the four-variable model diagram. We use this diagram to reduce notational verbosity. It is easy to see that R stands for REQ and NAT (the relationship between these two relations will be clarified in Chapter 4), P for IN, Q for OUT, and Z for SOF.

The existence conditions of the dotted arrows are obtained in a demonic semantics. The diagonal AC in the diagram is a demonic left factor of R through Q. The diagonal BD is a demonic right factor of R through P. In Sections 3.1 and 3.2 we prove necessary and sufficient existence conditions for

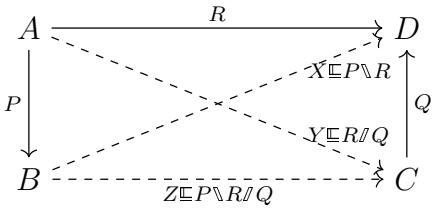


Figure 3.1: Demonic factorization

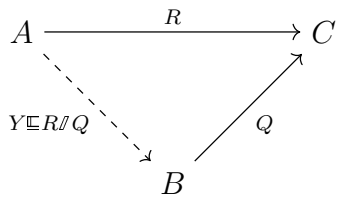
demonic left and, respectively, right factors. Using these conditions we will then prove in Section 3.3 a necessary and sufficient condition for the existence of a relation Z such that the diagram commutes. These results will be applied to the four-variable model in the subsequent chapters of the thesis.

3.1 Existence of a Demonic Left Factor

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 3.1. *Given two relations $R \subseteq A \times C$ and $Q \subseteq B \times C$, there exists a demonic left factor $Y \subseteq A \times B$ such that $Y \mathbin{\square} Q \sqsubseteq R$ if and only if*

$$\forall a \in \mathsf{dom}\,(R) \mathbin{\bullet} \exists b \in \mathsf{dom}\,(Q) \mathbin{\bullet} Q(b) \subseteq R(a)\,.$$



Proof.
If direction:

$$\begin{aligned}
&\exists Y \mathbin{\bullet} Y \mathbin{\square} Q \sqsubseteq R \\
\Rightarrow\ &\langle \text{by Definition 2.18, } R \mathbin{/\!\!/} Q \text{ is a solution of } Y \mathbin{\square} Q \sqsubseteq R \rangle \\
&(R \mathbin{/\!\!/} Q) \mathbin{\square} Q \sqsubseteq R \\
\Rightarrow\ &\langle \text{by Definition 2.14(i) of } \sqsubseteq \rangle \\
&\mathsf{dom}\,(R) \subseteq \mathsf{dom}\,((R \mathbin{/\!\!/} Q) \mathbin{\square} Q) \\
\Rightarrow\ &\mathsf{dom}\,(R) \subseteq \mathsf{dom}\,(R \mathbin{/\!\!/} Q) \\
\Rightarrow\ &\forall a \in \mathsf{dom}\,(R) \mathbin{\bullet} \exists b \in B \mathbin{\bullet} (a, b) \in R \mathbin{/\!\!/} Q \\
\Rightarrow\ &\langle \text{by (2.7)} \rangle
\end{aligned}$$

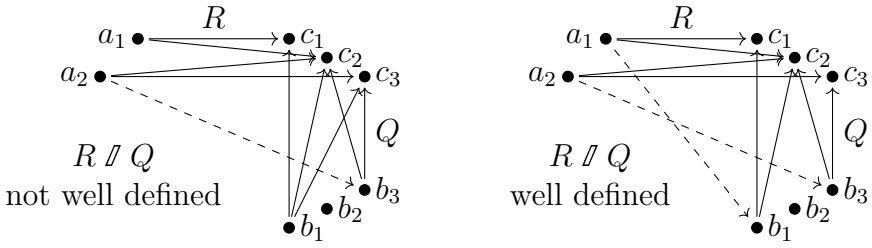
$$\forall a \in \mathsf{dom}\,(R) \,\boldsymbol{.}\, \exists b \in \mathsf{dom}\,(Q) \,\boldsymbol{.}\, Q(b) \subseteq R(a)\,.$$

Only if direction:

$$\begin{aligned}
&\forall a \in \mathsf{dom}\,(R) \,\boldsymbol{.}\, \exists b \in \mathsf{dom}\,(Q) \,\boldsymbol{.}\, Q(b) \subseteq R(a) \\
\Leftrightarrow\ &\langle \text{by (2.7)} \rangle \\
&\forall a \in \mathsf{dom}\,(R) \,\boldsymbol{.}\, \exists b \in \mathsf{dom}\,(Q) \,\boldsymbol{.}\, (a,b) \in R \mathbin{/\!\!/} Q \\
\Leftrightarrow\ &\forall a \in \mathsf{dom}\,(R) \,\boldsymbol{.}\, \exists b \in B \boldsymbol{.}\, (a,b) \in R \mathbin{/\!\!/} Q \wedge b \in \mathsf{dom}\,(Q) \\
\Leftrightarrow\ &\langle \text{by Definition 2.17 of } \triangledown \rangle \\
&\forall a \in \mathsf{dom}\,(R) \,\boldsymbol{.}\, \exists c \in C \boldsymbol{.}\, (a,c) \in (R \mathbin{/\!\!/} Q) \triangledown Q \\
\Rightarrow\ &\mathsf{dom}\,(R) \subseteq \mathsf{dom}\,((R \mathbin{/\!\!/} Q) \triangledown Q)\ . \qquad (3.1)
\end{aligned}$$

$$\begin{aligned}
&((R \mathbin{/\!\!/} Q) \triangledown Q)\,\big|_{\mathsf{dom}(R)} \subseteq R \\
\Leftrightarrow\ &\forall a \in \mathsf{dom}\,(R) \,\boldsymbol{.}\, ((R \mathbin{/\!\!/} Q) \triangledown Q)\,(a) \subseteq R(a) \\
\Leftrightarrow\ &\langle \text{by unfolding } \subseteq \rangle \\
&\forall a \in \mathsf{dom}\,(R) \,\boldsymbol{.}\, \forall c \in C \boldsymbol{.}\, (a,c) \in ((R \mathbin{/\!\!/} Q) \triangledown Q) \Rightarrow (a,c) \in R \\
\Leftrightarrow\ &\langle \text{by Definition 2.17 of } \triangledown \rangle \\
&\forall a \in \mathsf{dom}\,(R) \,\boldsymbol{.}\, \forall c \in C \boldsymbol{.}\, (\exists b \in \mathsf{dom}\,(Q) \,\boldsymbol{.}\, (a,b) \in R \mathbin{/\!\!/} Q \wedge (b,c) \in Q) \\
&\quad \Rightarrow (a,c) \in R \\
\Leftrightarrow\ &\langle \text{by (2.7)} \rangle \\
&\forall a \in \mathsf{dom}\,(R) \,\boldsymbol{.}\, \forall c \in C \boldsymbol{.}\, (a,c) \in R \Rightarrow (a,c) \in R \\
\Leftrightarrow\ &\mathsf{true}\,. \qquad (3.2)
\end{aligned}$$

$$\begin{aligned}
&\forall a \in \mathsf{dom}\,(R) \,\boldsymbol{.}\, \exists b \in \mathsf{dom}\,(Q) \,\boldsymbol{.}\, Q(b) \subseteq R(a) \\
\Rightarrow\ &\langle \text{by (3.1) \& (3.2) \& Definition 2.14 of } \sqsubseteq \rangle
\end{aligned}$$



(a) Lemma 3.1 not satisfied (b) Lemma 3.1 satisfied

Figure 3.2: Examples for the existence of a demonic left factor

$$
\begin{aligned}
& (R /\!\!/ Q) \mathbin{\square} Q \sqsubseteq R \\
\Rightarrow\ & \langle \text{by taking } X = R /\!\!/ Q \rangle \\
& \exists X \bullet X \mathbin{\square} Q \sqsubseteq R \,.
\end{aligned}
$$

■

The intuition for the necessary and sufficient condition given in Lemma 3.1 is that a demonic left factor of R through Q exists if and only if Q is at least as deterministic as R is, where the degree of (non)determinism of R and Q is given by the image sets of their inputs. The larger the image set of an input, the less deterministic the relation is for that input. This is illustrated in Figure 3.2, where we revisit an example from Section 2.3.2 regarding the definedness of the demonic left residual. There, we used the definition of the demonic left residual to verify its definedness. Here, we use the necessary and sufficient condition of Lemma 3.1. In Figure 3.2a the residual $R /\!\!/ Q$ is not well defined and a demonic left factor does not exist because for every input of R there is no input of Q whose image set in contained in the image set of that input of R. Thus, Q does not have the proper degree of determinism. In contrast, in Figure 3.2b the relation Q has the same degree of determinism at the output as R, hence a demonic left factor exists.

3.1.1 Comparison with Existence Conditions in the Literature

We now show that our necessary and sufficient condition for the existence of a demonic left factor given in Lemma 3.1 is indeed equivalent to the necessary and sufficient conditions (2.5) and (2.6) given in the literature.

Proposition 3.2. *Given two relations $R \subseteq A \times C$ and $Q \subseteq B \times C$,*

$$\forall a \in \mathsf{dom}\,(R) \mathbf{.} \exists b \in \mathsf{dom}\,(Q) \mathbf{.} Q(b) \subseteq R(a)$$

and

$$\mathsf{dom}\,(R) \subseteq \mathsf{dom}\,((R/Q) \fatsemi Q)$$

are equivalent.

Proof.

$$\begin{aligned}
&\forall a \in \mathsf{dom}\,(R) \mathbf{.} \exists b \in \mathsf{dom}\,(Q) \mathbf{.} Q(b) \subseteq R(a) \\
\Leftrightarrow\ &\langle \text{by (2.1)} \rangle \\
&\forall a \in \mathsf{dom}\,(R) \mathbf{.} \exists b \in \mathsf{dom}\,(Q) \mathbf{.} (a,b) \in R/Q \\
\Leftrightarrow\ &\forall a \in \mathsf{dom}\,(R) \mathbf{.} \exists b \in B \mathbf{.} (a,b) \in R/Q \wedge b \in \mathsf{dom}\,(Q) \\
\Leftrightarrow\ &\forall a \in \mathsf{dom}\,(R) \mathbf{.} \exists b \in B \mathbf{.} (a,b) \in R/Q \wedge \exists c \in C \mathbf{.} (b,c) \in Q \\
\Leftrightarrow\ &\forall a \in \mathsf{dom}\,(R) \mathbf{.} \exists c \in C \mathbf{.} \exists b \in B \mathbf{.} (a,b) \in R/Q \wedge (b,c) \in Q \\
\Leftrightarrow\ &\langle \text{by Definition 2.8 of } \fatsemi \rangle \\
&\forall a \in \mathsf{dom}\,(R) \mathbf{.} \exists c \in C \mathbf{.} (a,c) \in (R/Q) \fatsemi Q \\
\Leftrightarrow\ &\forall a \in \mathsf{dom}\,(R) \mathbf{.} a \in \mathsf{dom}\,((R/Q) \fatsemi Q) \\
\Leftrightarrow\ &\mathsf{dom}\,(R) \subseteq \mathsf{dom}\,((R/Q) \fatsemi Q)\ .
\end{aligned}$$

■

Proposition 3.3. *Given two relations $R \subseteq A \times C$ and $Q \subseteq B \times C$,*

$$\forall a \in \mathsf{dom}\,(R) \mathbin{\bullet} \exists b \in \mathsf{dom}\,(Q) \mathbin{\bullet} Q(b) \subseteq R(a)$$

and

$$\mathsf{dom}\,(R) \subseteq \mathsf{dom}\left((R/Q)\,\big|^{\mathsf{dom}(Q)}\right)$$

are equivalent.

Proof. It suffices to show that $\mathsf{dom}\left((R/Q)\,\big|^{\mathsf{dom}(Q)}\right) = \mathsf{dom}\,((R/Q) \fatsemi Q)$. Then, by Proposition 3.3, we will have a proof for the current statement.

From (2.7) we get:

$$\begin{aligned}(R/Q)\,\big|^{\mathsf{dom}(Q)} &= \{(a,b) \in A \times B \mid b \in \mathsf{dom}\,(Q) \wedge Q(b) \subseteq R(a)\} \\ &= \{(a,b) \in A \times B \mid (\exists c \in C \mathbin{\bullet} (b,c) \in Q) \wedge Q(b) \subseteq R(a)\}\,.\end{aligned}$$

Thus

$$\mathsf{dom}\left((R/Q)\,\big|^{\mathsf{dom}(Q)}\right) = \{a \in A \mid \exists b \in B \mathbin{\bullet} \exists c \in C \mathbin{\bullet} (b,c) \in Q \wedge Q(b) \subseteq R(a)\}\,.$$

We also have that

$$\begin{aligned}(R/Q) \fatsemi Q &= \{(a,c) \in A \times C \mid \exists b \in B \mathbin{\bullet} (a,b) \in R/Q \wedge (b,c) \in Q\} \\ &= \{(a,c) \in A \times C \mid \exists b \in B \mathbin{\bullet} Q(b) \subseteq R(a) \wedge (b,c) \in Q\}\,.\end{aligned}$$

Therefore

$$\mathsf{dom}\,((R/Q) \fatsemi Q) = \{a \in A \mid \exists c \in C \mathbin{\bullet} \exists b \in B \mathbin{\bullet} Q(b) \subseteq R(a) \wedge (b,c) \in Q\}\,.$$

Consequently, $\mathsf{dom}\left((R/Q)\,\big|^{\mathsf{dom}(Q)}\right) = \mathsf{dom}\,((R/Q) \fatsemi Q)$, which completes the proof. ■

The advantage of our necessary and sufficient condition given in Lemma 3.1 compared to the abstract relation algebraic conditions in the literature is a better insight into the constraints that the relations R and Q must satisfy in order

for a demonic left factor of R through Q to exist. Also, our condition is constructive in nature and suggests a procedure for "calculating" the demonic left residual $R \mathbin{/\!\!/} Q$: verifying the satisfiability of Lemma 3.1 by two relations R and Q requires the enumeration of all the pairs that satisfy the characteristic predicate of $R \mathbin{/\!\!/} Q$ as given in (2.7).

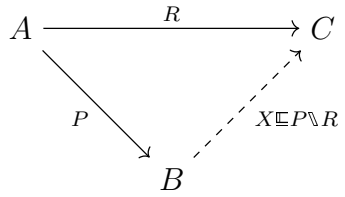
3.2 Existence of a Demonic Right Factor

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

Lemma 3.4. *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 $P \boxdot X \sqsubseteq R$ if and only if the following conditions are both satisfied:*

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

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



Proof.
If direction:

$$
\begin{aligned}
& \exists X \bullet P \boxdot X \sqsubseteq R \\
\Rightarrow\ & \langle \text{by Definition 2.19, } P \mathbin{\backslash\!\!\backslash} R \text{ is a solution of } P \boxdot X \sqsubseteq R \rangle \\
& P \boxdot (P \mathbin{\backslash\!\!\backslash} R) \sqsubseteq R \\
\Rightarrow\ & \langle \text{by Definition 2.14(i) of } \sqsubseteq \rangle \\
& \mathsf{dom}\,(R) \subseteq \mathsf{dom}\,(P \boxdot (P \mathbin{\backslash\!\!\backslash} R)) \\
\Rightarrow\ & \langle \text{by Definition 2.17 of } \boxdot \rangle
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{dom}\,(R) \subseteq \mathsf{dom}\,(P) \wedge \mathsf{ran}\left(P\big|_{\mathsf{dom}(R)}\right) \subseteq \mathsf{dom}\,(P \setminus\!\!\setminus R)\\
\Rightarrow\ &\mathsf{dom}\,(R) \subseteq \mathsf{dom}\,(P) \wedge \forall b \in \mathsf{ran}\left(P\big|_{\mathsf{dom}(R)}\right) \bullet \exists c \in C \bullet (b,c) \in P \setminus\!\!\setminus R\\
\Rightarrow\ &\langle \text{by (2.9)} \rangle\\
&\mathsf{dom}\,(R) \subseteq \mathsf{dom}\,(P)\\
&\wedge \forall b \in \mathsf{ran}\left(P\big|_{\mathsf{dom}(R)}\right) \bullet \exists c \in C \bullet \left(P\big|_{\mathsf{dom}(R)}\right)^{\smile}(b) \subseteq R^{\smile}(c)\,.
\end{aligned}
$$

Only if direction:

$$
\begin{aligned}
&\mathsf{dom}\,(R) \subseteq \mathsf{dom}\,(P)\\
&\wedge \forall b \in \mathsf{ran}\left(P\big|_{\mathsf{dom}(R)}\right) \bullet \exists c \in C \bullet \left(P\big|_{\mathsf{dom}(R)}\right)^{\smile}(b) \subseteq R^{\smile}(c)\\
\Rightarrow\ &\langle \text{by (2.9)} \rangle\\
&\mathsf{dom}\,(R) \subseteq \mathsf{dom}\,(P) \wedge \forall b \in \mathsf{ran}\left(P\big|_{\mathsf{dom}(R)}\right) \bullet \exists c \in C \bullet (b,c) \in P \setminus\!\!\setminus R\\
\Rightarrow\ &\forall a \in \mathsf{dom}\,(R) \bullet \forall b \in P(a) \bullet \exists c \in C \bullet (b,c) \in P \setminus\!\!\setminus R\\
\Rightarrow\ &\langle \text{by Definition 2.17 of } \lhd \rangle\\
&\forall a \in \mathsf{dom}\,(R) \bullet \exists c \in C \bullet (a,c) \in P \lhd (P \setminus\!\!\setminus R)\\
\Rightarrow\ &\mathsf{dom}\,(R) \subseteq \mathsf{dom}\,(P \lhd (P \setminus\!\!\setminus R))\ . \qquad (3.3)
\end{aligned}
$$

$$
\begin{aligned}
&(P \lhd (P \setminus\!\!\setminus R))\big|_{\mathsf{dom}(R)} \subseteq R\\
\Leftrightarrow\ &\forall a \in \mathsf{dom}\,(R) \bullet (P \lhd (P \setminus\!\!\setminus R))\,(a) \subseteq R(a)\\
\Leftrightarrow\ &\langle \text{by unfolding } \subseteq \rangle\\
&\forall a \in \mathsf{dom}\,(R) \bullet \forall c \in C \bullet (a,c) \in P \lhd (P \setminus\!\!\setminus R) \Rightarrow (a,c) \in R\\
\Leftrightarrow\ &\langle \text{by Definition 2.17 of } \lhd \rangle\\
&\forall a \in \mathsf{dom}\,(R) \bullet \forall c \in C \bullet (\exists b \in \mathsf{dom}\,(P \setminus\!\!\setminus R) \bullet (a,b) \in P \wedge (b,c) \in P \setminus\!\!\setminus R)\\
&\Rightarrow (a,c) \in R\\
\Leftrightarrow\ &\langle \text{by (2.9)} \rangle
\end{aligned}
$$

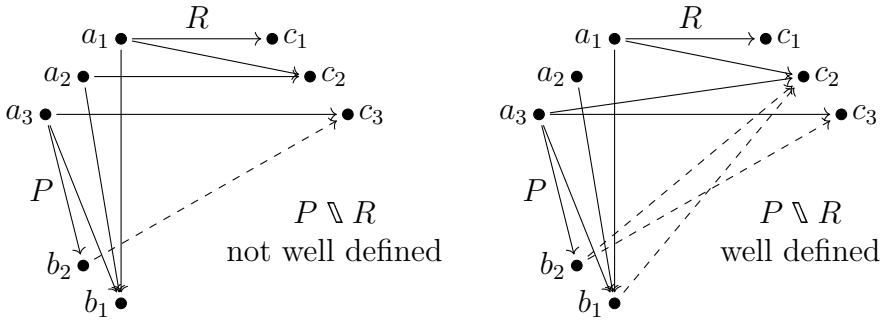
$$\forall a \in \mathsf{dom}\,(R) \bullet \forall c \in C \bullet (a, c) \in R \Rightarrow (a, c) \in R$$

$$\Leftrightarrow \mathsf{true}\,. \tag{3.4}$$

$$\begin{aligned}
&\mathsf{dom}\,(R) \subseteq \mathsf{dom}\,(P) \\
&\quad \wedge \forall b \in \mathsf{ran}\left(P\big|_{\mathsf{dom}(R)}\right) \bullet \exists c \in C \bullet \left(P\big|_{\mathsf{dom}(R)}\right)^{\smile}(b) \subseteq R^{\smile}(c) \\
&\Rightarrow \langle \text{by (3.3) \& (3.4) \& Definition 2.14 of } \sqsubseteq\!\!\!\sqsubseteq \rangle \\
&\Rightarrow P \rhd (P \setminus\!\!\setminus R) \sqsubseteq\!\!\!\sqsubseteq R \\
&\Rightarrow \langle \text{by taking } X = P \setminus\!\!\setminus R \rangle \\
&\quad \exists X \bullet P \rhd X \sqsubseteq\!\!\!\sqsubseteq R\,.
\end{aligned}$$

■

The intuition for the necessary and sufficient condition of Lemma 3.4 is that a demonic right factor of R through P exists if and only if P retains at least as much information about the inputs as R. The level of information that a relation preserves about its inputs is given by the reverse image sets of its outputs: the larger the reverse image set of an output, the higher the uncertainty of which inputs the output originates from. In Figure 3.3a, the reverse image set $\left(P\big|_{\mathsf{dom}(R)}\right)^{\smile}(b_1) = \{a_1, a_2, a_3\}$ is not contained in any reverse image set of an element in the range of R. This means that P is not able to distinguish between a_1, a_2, and a_3 when producing b_1, while R can make a distinction when producing c_1, c_2, or c_3. Therefore, in this example Lemma 3.4 is not satisfied; as such, a demonic right factor of R through P does not exist and $P \setminus\!\!\setminus R$ is not defined. In Figure 3.3b on the other hand, Lemma 3.4 is satisfied because $\left(P\big|_{\mathsf{dom}(R)}\right)^{\smile}(b_1) = \{a_1, a_3\} \subseteq R^{\smile}(c_2) = \{a_1, a_3\}$, hence a demonic right factor of R through P exists and $P \setminus\!\!\setminus R$ is defined. The same examples were used in Section 2.3.2 to illustrate the definedness of $P \setminus\!\!\setminus R$, but there the definition of the demonic right residual was used for checking the definedness of $P \setminus\!\!\setminus R$.



(a) Lemma 3.4 not satisfied (b) Lemma 3.4 satisfied

Figure 3.3: Examples for the existence of a demonic right factor

3.2.1 Comparison with Existence Conditions in the Literature

Our necessary and sufficient condition for the existence of a demonic right factor in Lemma 3.4 is in fact equivalent to the conditions given in (Frappier, 1995), (Desharnais et al., 1995), and (Kahl, 2003b). In Section 2.3.2 we showed that these conditions are equivalent to (2.8); to show the equivalence between our condition in Lemma 3.4 and (2.8), it suffices to prove the following statement.

Proposition 3.5. *Given two relations $R \subseteq A \times C$ and $P \subseteq A \times B$,*

$$\forall b \in \mathsf{ran}\left(P\big|_{\mathsf{dom}(R)}\right) \bullet \exists c \in C \bullet \left(P\big|_{\mathsf{dom}(R)}\right)^{\smile}(b) \subseteq R^{\smile}(c) \tag{3.5}$$

and

$$\mathbb{T}_{B,C} \subseteq \left(P\big|_{\mathsf{dom}(R)} \Big\backslash R\right) ; \mathbb{T}_{C,C} \tag{3.6}$$

are equivalent.

Proof. We start by noting that the set B can be written as the following union:

$$B = \mathsf{ran}\left(P\big|_{\mathsf{dom}(R)}\right) \cup \overline{\mathsf{ran}\left(P\big|_{\mathsf{dom}(R)}\right)}. \tag{3.7}$$

Specializing (2.2) to the context of this proof, we get

$$P\big|_{\mathsf{dom}(R)} \Big\backslash R = \left\{ (b,c) \in B \times C \;\middle|\; \left(P\big|_{\mathsf{dom}(R)}\right)^{\smile}(b) \subseteq R^{\smile}(c) \right\}. \tag{3.8}$$

Also, (3.6) is equivalent to $P\big|_{\mathsf{dom}(R)} \Big\backslash R$ being total.

If direction ((3.5) $\Rightarrow$ (3.6)):

This is a proof by cases. First case, for any $b \in \mathsf{ran}\left(P\big|_{\mathsf{dom}(R)}\right)$ we have by (3.5) that there exists a $c' \in C$ such that $\left(P\big|_{\mathsf{dom}(R)}\right)^{\smile}(b) \subseteq R^{\smile}(c')$, which by (2.9) means that $(b,c') \in P\big|_{\mathsf{dom}(R)} \Big\backslash R$. Second case, for any b in the complement of $\mathsf{ran}\left(P\big|_{\mathsf{dom}(R)}\right)$ the set $\left(P\big|_{\mathsf{dom}(R)}\right)^{\smile}(b) = \varnothing$, thus the characteristic predicate of $P\big|_{\mathsf{dom}(R)} \Big\backslash R$ in (3.8) is trivially satisfied. Consequently, $(b,c'') \in P\big|_{\mathsf{dom}(R)} \Big\backslash R$ for any $c'' \in C$. Considering these two cases and (3.7), we have that $P\big|_{\mathsf{dom}(R)} \Big\backslash R$ is total.

Only if direction ((3.6) $\Rightarrow$ (3.5)):

From (3.6) it follows that for any $b \in B$ there is a $c \in C$ such that $(b,c) \in P\big|_{\mathsf{dom}(R)} \Big\backslash R$. Then by (3.7) and (3.8) it follows that (3.5) holds. ■

Similarly to the case of demonic left factors, our condition for the existence of a demonic right factor given in Lemma 3.4 offers better insight into the constraints that the relations P and R must satisfy in order for a demonic right factor of R through P to exist. Moreover, our condition suggests a method to construct the demonic right residual $P \mathbin{\backslash\!\!\backslash} R$: by checking that for each element in the range of $P\big|_{\mathsf{dom}(R)}$ there exists an element in the range of R that satisfies Lemma 3.4(ii), we essentially enumerate all the pairs that satisfy the characteristic predicate of $P \mathbin{\backslash\!\!\backslash} R$ as given in (2.9).

3.3 Existence of a Demonic Mid Factor

Ultimately, we are interested in a necessary and sufficient condition for the existence of a relation Z in Figure 3.1 such that $P \rhd Z \rhd Q \sqsubseteq R$.

Definition 3.6. A demonic mid factor of $R \subseteq A \times D$ through $P \subseteq A \times B$ and $Q \subseteq C \times D$ is a relation $Z \subseteq B \times C$ such that $P \rhd Z \rhd Q \sqsubseteq R$.

Demonic composition is an associative operation (Backhouse and van der Woude, 1993), that is, $P \rhd (Z \rhd Q) = (P \rhd Z) \rhd Q$. The associativity of $\rhd$ implies that both diagonals X and Y in Figure 3.1 are necessary for Z to be a demonic mid factor of R through P and Q since we can take $X = Z \rhd Q$ and $Y = P \rhd Z$. The diagonals X and Y are demonic right and, respectively, left factors of R. A necessary and sufficient condition for X to exist is given by applying Lemma 3.4 in triangle A, B, D:

$$\begin{aligned} &\mathsf{dom}\,(R) \subseteq \mathsf{dom}\,(P) \\ &\wedge\, \forall b \in \mathsf{ran}\left(P\big|_{\mathsf{dom}(R)}\right) \bullet \exists d \in D \bullet \left(P\Big|_{\mathsf{dom}(R)}\right)^{\smile}(b) \subseteq R^{\smile}(d)\,. \end{aligned} \tag{3.9}$$

By Definition 2.19, the largest X with respect to $\sqsubseteq$ is the demonic right residual $P \mathbin{\backslash\!\!\backslash} R$. Similarly, a necessary and sufficient condition for Y to exist is given by Lemma 3.1 applied in triangle A, C, D:

$$\forall a \in \mathsf{dom}\,(R) \bullet \exists c \in \mathsf{dom}\,(Q) \bullet Q(c) \subseteq R(a)\,. \tag{3.10}$$

By Definition 2.18, the largest Y with respect to $\sqsubseteq$ is the demonic left residual $R \mathbin{/\!\!/} Q$.

The existence of the diagonals X and Y, however, is not sufficient for a demonic mid factor Z to exist. A counterexample to the sufficiency of their conjunction is provided in Figure 3.4. In this example, (3.9) 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 (3.10) 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 $(b_1, c_1) \in Z$, then a_2 can be connected to d_1 via $P \rhd Z \rhd Q$ although $(a_2, d_1) \notin R$; similarly, if $(b_1, c_3) \in Z$,

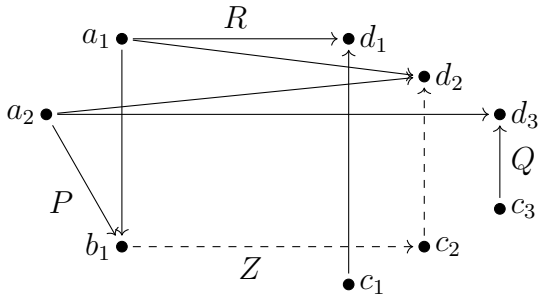


Figure 3.4: The diagonals are not sufficient for a demonic mid factor

then a_1 can reach d_3 via $P \boxdot Z \boxdot Q$ although $(a_1, d_3) \notin R$. Consequently, there is no relation Z such that $P \boxdot Z \boxdot Q \sqsubseteq R$, although both (3.9) and (3.10) are satisfied. It is only when $(c_2, d_2) \in Q$ that there is a $Z = \{(b_1, c_2)\}$ such that $P \boxdot Z \boxdot Q \sqsubseteq R$. It can be seen in Figure 3.4 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 Z to exist.

Hence, we give the following necessary and sufficient condition for the existence of a demonic mid factor Z.

Lemma 3.7. *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 $Z \subseteq B \times C$ such that $P \boxdot Z \boxdot Q \sqsubseteq R$ if and only if the following conditions are both satisfied:*

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

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

Proof. The geometrical interpretation in Figure 3.4 of the associativity of $\boxdot$ 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 Z. As such, it suffices to use the

diagonal BD and show that the conditions in Lemma 3.7 are necessary and sufficient for Z such that $P \mathbin{\square} (Z \mathbin{\square} Q) \sqsubseteq R$.

$$
\begin{aligned}
& \exists Z \mathbin{\bullet} P \mathbin{\square} (Z \mathbin{\square} Q) \sqsubseteq R \\
\Leftrightarrow\ & \langle \text{by Definition 2.19 of } \mathbin{\backslash\!\!\backslash} \text{ \& Lemma 3.4 applied in } \triangle A, B, D \rangle \\
& (\exists Z \mathbin{\bullet} Z \mathbin{\square} Q \sqsubseteq P \mathbin{\backslash\!\!\backslash} R) \wedge (3.9) \\
\Leftrightarrow\ & \langle \text{by Lemma 3.1 applied in } \triangle B, C, D \rangle \\
& (\forall b \in \mathsf{dom}\,(P \mathbin{\backslash\!\!\backslash} R) \mathbin{\bullet} \exists c \in \mathsf{dom}\,(Q) \mathbin{\bullet} Q(c) \subseteq (P \mathbin{\backslash\!\!\backslash} R)(b)) \wedge (3.9) \\
\Leftrightarrow\ & \left\langle \mathsf{dom}\,(P \mathbin{\backslash\!\!\backslash} R) = \mathsf{ran}\left(P\big|_{\mathsf{dom}(R)}\right) \right. \\
& \quad \left. \& \ (P \mathbin{\backslash\!\!\backslash} R)(b) = \left\{ d \in D \;\middle|\; \left(P\big|_{\mathsf{dom}(R)}\right)^{\smile}(b) \subseteq R^{\smile}(d) \right\} \right\rangle \\
& \mathsf{dom}\,(R) \subseteq \mathsf{dom}\,(P) \\
& \wedge \forall b \in \mathsf{ran}\left(P\big|_{\mathsf{dom}(R)}\right) \mathbin{\bullet} \\
& \quad \exists c \in \mathsf{dom}\,(Q) \mathbin{\bullet} Q(c) \subseteq \left\{ d \in D \;\middle|\; \left(P\big|_{\mathsf{dom}(R)}\right)^{\smile}(b) \subseteq R^{\smile}(d) \right\}.
\end{aligned}
$$

■

An alternative formulation of the necessary and sufficient condition for the existence of a demonic mid factor is given in the following theorem. This form is preferred in the sequel since it is notationally more convenient than Lemma 3.7.

Theorem 3.8. *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 $Z \subseteq B \times C$ such that $P \mathbin{\square} Z \mathbin{\square} Q \sqsubseteq R$ if and only if the following conditions are both satisfied:*

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

(ii) $\forall b \in \mathsf{ran}\left(P\big|_{\mathsf{dom}(R)}\right) \mathbin{\bullet} \exists c \in \mathsf{dom}\,(Q) \mathbin{\bullet} Q(c) \subseteq \bigcap_{a \in A'} R(a)$*, where* $A' = \left(P\big|_{\mathsf{dom}(R)}\right)^{\smile}(b)$*.*

Proof. Let $D' = \left\{ d \in D \;\middle|\; \left(P\big|_{\mathsf{dom}(R)}\right)^{\smile}(b) \subseteq R^{\smile}(d) \right\}$ and $A' = \left(P\big|_{\mathsf{dom}(R)}\right)^{\smile}(b)$, for any $b \in \mathsf{ran}\left(P\big|_{\mathsf{dom}(R)}\right)$.

First, we prove that $D' \subseteq \bigcap_{a \in A'} R(a)$. For this, we show that for any $d \in D'$ it is also the case that $d \in \bigcap_{a \in A'} R(a)$:

$$
\begin{aligned}
&\forall d \in D \mathbf{.}\, d \in D' \Rightarrow A' \subseteq R^{\smile}(d) \\
\Rightarrow &\forall d \in D \mathbf{.}\, d \in D' \Rightarrow \forall a \in A' \mathbf{.}\, a \in R^{\smile}(d) \\
\Rightarrow &\forall d \in D \mathbf{.}\, d \in D' \Rightarrow \forall a \in A' \mathbf{.}\, (a, d) \in R \\
\Rightarrow &\forall d \in D \mathbf{.}\, d \in D' \Rightarrow d \in \bigcap_{a \in A'} R(a)\,.
\end{aligned}
$$

Second, we prove that $\bigcap_{a \in A'} R(a) \subseteq D'$. For this, we show that for any $d \in \bigcap_{a \in A'} R(a)$ it is also the case that $d \in D'$:

$$
\begin{aligned}
&\forall d \in D \mathbf{.}\, d \in \bigcap_{a \in A'} R(a) \Rightarrow \forall a \in A' \mathbf{.}\, (a, d) \in R \\
\Rightarrow &\forall d \in D \mathbf{.}\, d \in \bigcap_{a \in A'} R(a) \Rightarrow \forall a \in A' \mathbf{.}\, a \in R^{\smile}(d) \\
\Rightarrow &\forall d \in D \mathbf{.}\, d \in \bigcap_{a \in A'} R(a) \Rightarrow A' \subseteq R^{\smile}(d) \\
\Rightarrow &\forall d \in D \mathbf{.}\, d \in \bigcap_{a \in A'} R(a) \Rightarrow d \in D'\,.
\end{aligned}
$$

Consequently, $D' = \bigcap_{a \in A'} R(a)$, and, by Lemma 3.7, we have a proof for Theorem 3.8. ■

Similarly to the demonic left and right residuals, our demonic mid residual is the largest demonic mid factor.

Theorem 3.9. *Given relations $R \subseteq A \times D$, $P \subseteq A \times B$, $Z \subseteq B \times C$, and $Q \subseteq C \times D$, if Z is a demonic mid factor of R through P and Q, then Z is a demonic refinement of the residual $P \setminus\!\!\setminus R \mathbin{/\!\!/} Q$:*

$$P \mathbin{\square} Z \mathbin{\square} Q \sqsubseteq R \Rightarrow Z \sqsubseteq P \setminus\!\!\setminus R \mathbin{/\!\!/} Q\,.$$

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

An implication of Theorem 3.9 is that the residual $P \setminus\!\!\setminus R /\!\!/ Q$ is the largest solution, with respect to $\sqsubseteq$, of the inequation $P \sqcap Z \sqcap Q \sqsubseteq R$. We call this residual the *demonic mid residual* of R by P and Q. By Theorem 3.8 and Theorem 3.9, the demonic mid residual is defined only when a demonic mid factor exists. Theorem 3.8 also gives us the value of the demonic mid residual:

$$P \setminus\!\!\setminus R /\!\!/ Q \rightrightarrows \left\{ (b,c) \in B \times C \;\middle|\; b \in \mathsf{ran}\left(P\big|_{\mathsf{dom}(R)}\right) \wedge c \in \mathsf{dom}\,(Q) \right.$$
$$\left. \wedge\, Q(c) \subseteq \bigcap_{a \in A'} R(a) \right\},$$
$$\text{where } A' = \left(P\big|_{\mathsf{dom}(R)}\right)^{\smile}(b)\,. \tag{3.11}$$

As was the case with the demonic left and right factors, the condition for the existence of a demonic mid factor given in Theorem 3.8 has a constructive flavour. Checking the existence of a demonic mid factor of R through P and Q requires checking that for each element in the range of $P\big|_{\mathsf{dom}(R)}$ there exists an element in the domain of Q that satisfies Theorem 3.8(ii). This essentially reduces to enumerating all the pairs in the demonic mid residual $P \setminus\!\!\setminus R /\!\!/ Q$.

3.4 Summary

In this chapter we have used a diagram isomorphic to the diagram of the four-variable model to reduce notational verbosity, but also to allow a more straightforward comparison with the results from the relation algebra literature. To answer the question of implementability of system requirements, we

are interested in existence conditions for the diagonals of the diagram as well as for the relation that corresponds to *SOF*, such that the diagram commutes. The condition for the diagram to commute is demonic refinement, while the sequential composition of the relations in the diagram is demonic composition. The rationale for choosing demonic refinement and composition will be further explained in Chapter 4.

Two contributions presented in this chapter are the necessary and sufficient conditions for the existence of demonic left and, respectively, right factors. These conditions were used to derive existence conditions for the diagonals and for the relation corresponding to *SOF*. Although the conditions turned out to be equivalent to the abstract algebraic conditions presented in the literature, we believe that they are more accesible for an engineering audience since they are given in predicate logic. The conditions also offer a better insight into what is required for demonic left and right factors to exist. This insight will be explored further in Chapter 5 where the existence of the diagonals will be used to define the notions of observability and controllability of the system requirements with respect to the input and, respectively, output hardware interfaces.

The main contribution of the chapter is a necessary and sufficient condition for the existence of the relation that corresponds to *SOF*. We call this relation a demonic mid factor of the relation that corresponds to *REQ* and *NAT* (the relationship between these two relations will be clarified in Chapter 4) through the relations that correspond to *IN* and *OUT*. This result is to the best of our knowledge new in relation algebra.

The necessary and sufficient existence conditions for the demonic left, right, and mid factors presented in this chapter are constructive because checking for the existence of the factors requires finding all the pairs that satisfy the characteristic predicates of the corresponding demonic residuals. A method to "calculate" the demonic residuals is useful since they are the weakest (i.e., least restrictive) specifications or implementation descriptions that are still acceptable without placing unnecessary constraints on actual implementations.

Chapter 4

Implementability of System Requirements

In this chapter we answer the main question of the thesis and give a necessary and sufficient existence condition for an acceptable software specification, given the constraints imposed on the software by the environment, system requirements, and input/output devices. In Section 4.1 we describe the shortcomings of the angelic acceptability criterion proposed by (Parnas and Madey, 1995). In particular, this criterion allows system specifications that are not completely consistent with the natural laws of the environment. As a first step to address these issues, in Section 4.2 we redefine in the demonic calculus of relations the notion of feasibility of system requirements proposed by Parnas and Madey so as the system requirements specify for every input possible in the environment only outputs allowed by the environment. As a second step, in Section 4.3 we redefine system and software acceptability in a demonic setting such that it does not allow nonterminating or empty implementations. Then in Section 4.4 we give a necessary and sufficient condition for the existence of an acceptable software specification. Our approach also yields a formal characterization of the software requirements, which is the subject of Section 4.5.

4.1 The Angelic Acceptability Notion of Parnas and Madey

From an engineering perspective, an implementation is not possible if a behaviour specified in the requirements is not physically meaningful. 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 and may even be dangerous as they can overstress the engines and airframe. The requirements should also specify the response expected from the system for all values allowed by the physical environment for the monitored variables. If, given a physical environment NAT, system requirements REQ satisfy the two properties mentioned above, then REQ is said to be *feasible* with respect to NAT. The feasibility property is formalized in (Parnas and Madey, 1995) using the following conditions:

$$\mathsf{dom}\,(NAT) \subseteq \mathsf{dom}\,(REQ) \tag{4.1}$$

and

$$\mathsf{dom}\,(REQ \cap NAT) = \mathsf{dom}\,(REQ) \cap \mathsf{dom}\,(NAT)\ . \tag{4.2}$$

Condition (4.1) asks REQ to specify system response for all the values of monitored variables that can arise in the environment. It can be assumed that the values not contained in the domain of NAT will never occur under normal environmental circumstances. Condition (4.2) says that for each input they have in common, REQ and NAT should agree on at least one output. Together, conditions (4.1) and (4.2) ensure that, for every input allowed by the environment, the system requirements ask the system to produce at least one output that is physically meaningful.

Acceptability of software is defined in (Parnas and Madey, 1995) as follows:

$$NAT \cap (IN\ ;\ SOF\ ;\ OUT) \subseteq REQ\,. \tag{4.3}$$

A system implementation $SYS = IN\ ;\ SOF\ ;\ OUT$ is then acceptable if and

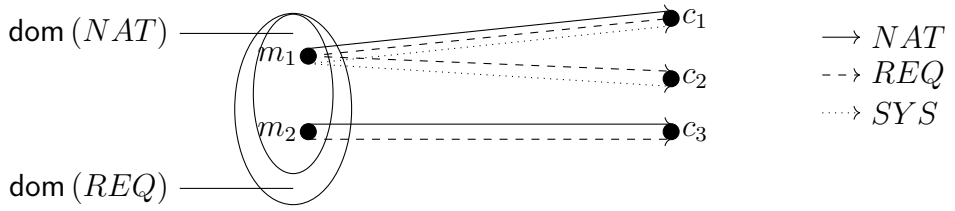


Figure 4.1: The acceptability conditions of Parnas and Madey are too weak

only if it satisfies the following condition:

$$NAT \cap SYS \subseteq REQ\,. \tag{4.4}$$

These acceptability conditions are, however, not strong enough. Let us consider the relations NAT, REQ, and SYS in Figure 4.1, where m_1, m_2 are values of monitored variables and c_1, c_2, c_3 are values of controlled variables. These relations satisfy the conditions (4.1), (4.2) and (4.4). Therefore, according to (Parnas and Madey, 1995), the system requirements REQ are feasible with respect to NAT and the system implementation SYS and software SOF are acceptable although:

- $(m_2, c_3) \in NAT$, $(m_2, c_3) \in REQ$, and $(m_2, c_3) \notin SYS$. That is, a system implementation SYS that does not deal with all the inputs in $\mathsf{dom}\,(NAT)$ and $\mathsf{dom}\,(REQ)$ is deemed acceptable. It is mentioned elsewhere in (Parnas and Madey, 1995) that $\mathsf{dom}\,(NAT) \subseteq \mathsf{dom}\,(IN)$; this, however, does not ensure that $\mathsf{dom}\,(NAT) \subseteq \mathsf{dom}\,(IN\,;SOF\,;OUT)$.

- $(m_1, c_2) \in SYS$ and $(m_1, c_2) \notin NAT$. That is, a system specification SYS that asks its implementations to produce outputs not physically possible is deemed acceptable. From an engineering perspective, such implementations are not realizable and it is important to reject early specifications that allow them. A similar problem with the acceptability condition in (Parnas and Madey, 1995) was pointed out by (Gunter et al., 2000). In Section 4.6 we will use the example from (Gunter et al., 2000) to show how our demonic approach remedies this problem.

  The reason why the acceptability conditions (4.3) and (4.4) of (Parnas

and Madey, 1995) allow implementation specifications (descriptions) such as the one in Figure 4.1 despite the problems described above is due to the relational intersection and inclusion in these conditions. If an implementation is not completely consistent with NAT (i.e., it has inputs not in the domain of NAT, or outputs not in the range of NAT, or both), then its intersection with NAT is still contained in REQ if the part of the implementation that obeys NAT does not violate REQ. Two extreme cases are when an implementation is completely inconsistent with NAT or when an implementation is empty, in both cases the intersection with NAT being empty and the inclusion in REQ being trivially satisfied.

Another problem with the acceptability conditions (4.3) and (4.4) of (Parnas and Madey, 1995) is that they allow nonterminating implementations. This happens when $\mathsf{dom}\,(IN) \supseteq \mathsf{dom}\,(SOF)$ or $\mathsf{dom}\,(SOF) \supseteq \mathsf{dom}\,(OUT)$, and an implementation $IN\,;SOF\,;OUT$ gets stuck in between IN and SOF or in between SOF and OUT, resulting in a nondeterministic behaviour which, for some inputs, sometimes produces expected outputs and some other times does not produce any results at all. As discussed in Section 1.3.1, such nondeterminism is angelic and ensures partial correctness only.

We will address the issues described in this section in the remainder of the chapter using the demonic calculus of relations.

4.2 Feasibility of System Requirements

The first step in our attempt to remedy the aforementioned shortcomings of the formalization by (Parnas and Madey, 1995) is to redefine feasibility of system requirements.

Revisiting Figure 4.1, 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 behaviours for the inputs allowed by NAT, then the following new

definition should be used for the feasibility of system requirements.

Definition 4.1. 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, 1995). The latter requires REQ to specify at least one output allowed by NAT for every input in the domain of NAT (conditions (4.1) and (4.2)). The feasibility in Definition 4.1, 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 4.1 is well defined.

Theorem 4.2. *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. However, because the demonic lattice is a join semilattice, the demonic meet $\sqcap$ does not always exist. Thus, we need to make sure that $\sqcap$ is well defined in either direction of $\Leftrightarrow$.

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

$$REQ = REQ\big|_{\mathsf{dom}(NAT)} \cup REQ\big|_{\overline{\mathsf{dom}(NAT)}}\,. \tag{4.5}$$

By (2.4) we also have that

$$REQ \sqcap NAT = (REQ \cap NAT) \cup REQ\big|_{\overline{\mathsf{dom}(NAT)}} \cup NAT\big|_{\overline{\mathsf{dom}(REQ)}}\,. \tag{4.6}$$

"$\Rightarrow$" direction:

Assuming $REQ = REQ \sqcap NAT$, it follows by (4.6) that

$$REQ = (REQ \cap NAT) \cup REQ\big|_{\overline{\mathsf{dom}(NAT)}} \cup NAT\big|_{\overline{\mathsf{dom}(REQ)}}\,. \tag{4.7}$$

By combining (4.5) and (4.7), and cancelling $REQ\big|_{\overline{\mathsf{dom}(NAT)}}$ on both sides, we get

$$REQ\big|_{\mathsf{dom}(NAT)} = (REQ \cap NAT) \cup NAT\big|_{\overline{\mathsf{dom}(REQ)}}. \tag{4.8}$$

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

$$\mathsf{dom}\,(NAT) \subseteq \mathsf{dom}\,(REQ)\ . \tag{4.9}$$

With $NAT\big|_{\overline{\mathsf{dom}(REQ)}}$ empty, (4.8) becomes

$$REQ\big|_{\mathsf{dom}(NAT)} = REQ \cap NAT\,. \tag{4.10}$$

Equation (4.10) 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 Definition 2.14 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 well defined. Because (4.9), we have that

$$\mathsf{dom}\,(REQ) \cap \mathsf{dom}\,(NAT) = \mathsf{dom}\,(NAT)\ . \tag{4.11}$$

By (4.10), $\mathsf{dom}\left(REQ\big|_{\mathsf{dom}(NAT)}\right) = \mathsf{dom}\,(REQ \cap NAT)$. Because (4.9), we also have that $\mathsf{dom}\left(REQ\big|_{\mathsf{dom}(NAT)}\right) = \mathsf{dom}\,(NAT)$. Consequently:

$$\mathsf{dom}\,(REQ \cap NAT) = \mathsf{dom}\,(NAT)\ . \tag{4.12}$$

By (4.11) and (4.12), the following equality holds:

$$\mathsf{dom}\,(REQ \cap NAT) = \mathsf{dom}\,(REQ) \cap \mathsf{dom}\,(NAT)\ . \tag{4.13}$$

This equality is exactly the same as the second condition of the feasibility notion proposed by Parnas and Madey (4.2), which is in fact equivalent to the compatibility condition (2.3) 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 Definition 2.14 of demonic refinement that $\mathsf{dom}\,(NAT)$ is contained in $\mathsf{dom}\,(REQ)$. This implies that

$$NAT\big|_{\overline{\mathsf{dom}(REQ)}} = \perp\!\!\!\perp\,. \tag{4.14}$$

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 (4.10) holds. From (4.5), (4.6), (4.10) and (4.14), it follows that $REQ = REQ \sqcap NAT$.

The definedness of the demonic intersection is proved the same way as in the "$\Rightarrow$" direction. ■

The check for feasibility of system requirements can be done as part of the requirements validation process. If the system 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.

4.3 System and Software Acceptability

We now redefine the angelic acceptability notion of (Parnas and Madey, 1995) in the demonic calculus of relations.

Definition 4.3. Given feasible system requirements REQ, a system implementation SYS is acceptable with respect to REQ and physical environment NAT if $SYS \sqsubseteq REQ \sqcap NAT$.

For an acceptable system implementation SYS, Theorem 4.2 ensures that $REQ \sqcap NAT$ is well defined and also that the following refinement ordering holds:

$$SYS \sqsubseteq REQ \sqsubseteq NAT\,.$$

Consequently, by Definition 2.14 of demonic refinement, 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 behaviour for abnormal circumstances when the environment is perturbed by phenomena that are independent of the system. Allowing arbitrary behaviour 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 behaviours have been added to the domain of REQ as additional safety requirements.

In (Parnas and Madey, 1995), a system implementation is given as $SYS = IN \,;\, SOF \,;\, OUT$. As seen in Section 2.3.2, angelic composition allows dead ends between the composed relations. Let us now consider a more concrete example that shows how an angelic semantics can lead to undesirable system behaviours. For example, let us consider a relation IN that models an 8-bit resolution ADC which converts monitored voltages m in the range 0–5V 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 ± 0.04V. Because the relation NAT says that the monitored voltages will be in the range 0–2.49V, 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 ± 0.02V, the following situation depicted in Figure 4.2 can occur: for $m = 2.49V$, 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, a result not allowed by REQ is produced). The angelic acceptability condition (4.3) of Parnas and Madey is trivially satisfied although an implementation $IN \,;\, SOF \,;\, OUT$ can produce an overflow: for example, for

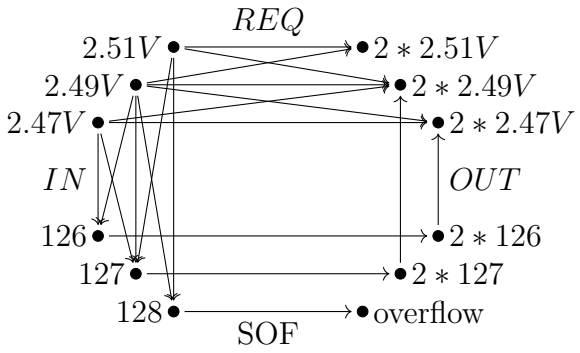


Figure 4.2: Angelic semantics allows undesirable implementations

$m = 2.49V$, $NAT \cap (IN \fatsemi SOF \fatsemi OUT) = \{(2.49V, 2 * 2.47V), (2.49V, 2 * 2.49V)\} \subseteq REQ = \{(2.49V, 2 * 2.47V), (2.49V, 2 * 2.49V), (2.49V, 2 * 2.51V)\}$. If we redefine the system implementation using demonic composition, then $SYS = IN \fatsemi SOF \fatsemi OUT = \perp\!\!\!\perp$ for $m = 2.49V$; this system implementation will not demonically refine a non-empty $REQ \sqcap NAT$ and, by Definition 4.3, it 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 4.4. Given feasible system requirements REQ, a software implementation SOF is acceptable with respect to REQ, input interface IN, output interface OUT, and physical environment NAT if SOF is a demonic mid factor of $REQ \sqcap NAT$ through IN and OUT:

$$IN \fatsemi SOF \fatsemi OUT \sqsubseteq REQ \sqcap NAT\,.$$

4.4 Existence of Acceptable Software

When designing a system, a difficult task is to find the right triple IN, OUT, and SOF such that their integration produces an acceptable system design. Usually, the system designers specify IN and OUT and the software engineers need to figure out an acceptable SOF. In such cases and assuming that the

system requirements are feasible, implementability of requirements reduces to the existence of an acceptable software specification.

The mathematical question we ask is, given relations NAT, REQ, IN, and OUT, does a relation SOF exist such that $IN \boxdot SOF \boxdot OUT \sqsubseteq REQ \pitchfork NAT$? The following theorem answers this question.

Theorem 4.5. *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) $\mathsf{dom}\,(REQ \pitchfork NAT) \subseteq \mathsf{dom}\,(IN)$*;*

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

$$OUT(o) \subseteq \bigcap_{m \in M'} (REQ \pitchfork NAT)\,(m)\,,$$

where $M' = \left(IN\big|_{\mathsf{dom}(REQ \pitchfork NAT)}\right)^{\smile}(i)$.

Proof. An acceptable SOF is a demonic mid factor of $REQ \pitchfork NAT$ through IN and OUT. As such, the current theorem is a direct consequence of the necessary and sufficient existence condition for a demonic mid factor given in Theorem 3.8. ■

4.5 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" (Miller and Tribble, 2001), hence an automated method would be 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.

Following from Definition 4.4 and Theorem 3.9, we have that any acceptable software, if it exists, is a demonic refinement of the demonic mid residual $IN \setminus\!\setminus (REQ \pitchfork NAT) \mathbin{/\!\!/} OUT$. As a result, this residual is the least restrictive software specification, or the *weakest software specification*, as it leaves open most software design options. The weakest software specification describes all the possible acceptable software implementations and, in this sense, it describes the software requirements.

Definition 4.6. 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 $REQ \pitchfork NAT$ through IN and OUT:

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

The software requirements are well defined only when an acceptable SOF exists. A program that satisfies the software requirements is a functional (i.e., deterministic) demonic refinement of SOF_{req}. This suggests a possible software development process based on stepwise refinement, where SOF_{req} is a sound starting point for the software design process. In such a process, a software design and, eventually, a program are guaranteed to be acceptable by construction if demonic refinement is preserved.

If the software requirements are well defined, then they can, in principle, be derived by "calculating" the value of the residual $IN \setminus\!\setminus (REQ \pitchfork NAT) \mathbin{/\!\!/} OUT$. One way to calculate this demonic residual is to use its abstract relation-algebraic value. 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. (Schmidt and Ströhlein, 1993; Schmidt, 2011). RelView[1], with its library Kure2[2], is a tool that supports the manipulation of relations represented as Boolean matrices using an

[1] `http://www.informatik.uni-kiel.de/~progsys/relview/`
[2] `http://www.informatik.uni-kiel.de/~progsys/kure2/`

optimized implementation based on binary decision diagrams. Another way to calculate the residual $IN \setminus\!\setminus (REQ \rightthreetimes NAT) \mathbin{/\!\!/} OUT$ is to use its concrete value given by formula (3.11):

$$SOF_{req} \backsimeq \left\{ (i,o) \in \mathbf{I} \times \mathbf{O} \;\middle|\; i \in \mathsf{ran}\left(IN\big|_{\mathsf{dom}(REQ \rightthreetimes NAT)}\right) \wedge o \in \mathsf{dom}\,(OUT) \right.$$
$$\left. \wedge\, OUT(o) \subseteq \bigcap_{m \in M'} (REQ \rightthreetimes NAT)\,(m) \right\},$$
$$\text{where } M' = \left(IN\big|_{\mathsf{dom}(REQ \rightthreetimes NAT)}\right)^{\smile}(i)\,. \tag{4.15}$$

When calculating the software requirements is not feasible for very large relations, or in the case of infinite relations, reasoning about relational specifications is still possible in an interactive proof assistant such as Coq, Isabelle, or PVS.

Techniques for deriving the software requirements from the specifications of system requirements and input/output hardware interfaces in the four-variable model are mentioned in (Thompson et al., 1999), (Thompson et al., 2000), (Bharadwaj and Heitmeyer, 2000), (Heimdahl and Thompson, 2000), (Lawford et al., 2000), (Miller and Tribble, 2001), (Wassyng and Lawford, 2003) and (Wassyng and Lawford, 2006). These techniques are variations of the same idea of manually decomposing *SOF* into three subrelations, as illustrated in Figure 4.3. The relations SOF_{in} and SOF_{out} model the input and, respectively, output device drivers and create the software approximations of the monitored and, respectively, controlled variables. The relation *REQ'* from the set *M'* of software approximations of monitored variables to the set *C'* of software approximations of controlled variables, closely resembles *REQ* and is regarded as the software requirements for the control software [3]. The idea is based on Parnas' information hiding principle (Parnas, 1972) to prevent local changes from propagating elsewhere in the system: *REQ'* is isolated from changes in the input and output devices, which will affect only the drivers

[3]The relation *REQ'* is called "pseudo requirements", while *M'* and *C'* are called the sets of "pseudo" monitored and, respectively, controlled variables in (Lawford et al., 2000; Wassyng and Lawford, 2003; Wassyng and Lawford, 2006).

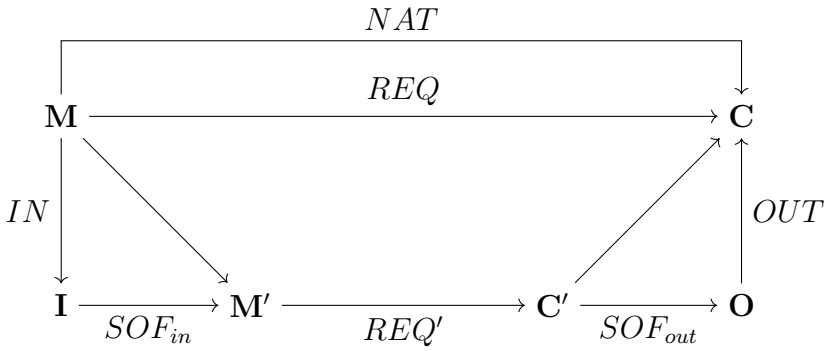


Figure 4.3: Isolation of input/output hardware drivers

SOF_{in} and SOF_{out}. Moreover, if the system requirements *REQ* change, *REQ'* will, in principle, be straightforward to derive. We do not decompose *SOF* in this dissertation, however, the results presented readily apply if we choose to do so.

4.6 Examples

In this section we revisit the abstract example we gave in Figure 4.1 and show how our demonic approach solves the shortcomings of the angelic acceptability notion proposed by (Parnas and Madey, 1995). In the same vein, we show how our approach works on the example given in (Gunter et al., 2000).

Example 1

Our abstract example in Figure 4.1 highlights two main problems with the acceptability proposed in (Parnas and Madey, 1995).

The first problem is that (m_2, c_3) belongs to both *NAT* and *REQ*, but not to *SYS*. Therefore, implementations that do not deal with all the inputs possible from the environment *NAT* are considered acceptable. We tackle this issue with our definition of feasibility of *REQ* with respect to *NAT* and our definitions of acceptability (Definitions 4.3 and 4.4). Together, feasibility of *REQ* and acceptability of an implementation *SYS* ensure the following refinement ordering: $SYS \sqsubseteq REQ \sqsubseteq NAT$, which requires $\mathsf{dom}\,(NAT) \subseteq$

$\mathsf{dom}\,(REQ) \subseteq \mathsf{dom}\,(SYS)$. This does not allow pairs (m_2, c_3) that belongs to NAT and REQ, but not to SYS.

The second problem concerns system specifications (descriptions) that are allowed to have outputs not physically possible. In Figure 4.1 this case was depicted by the pair (m_1, c_2) that belongs to SYS, but no to NAT. Again, our definitions for feasibility and acceptability imply that $SYS \sqsubseteq REQ \sqsubseteq NAT$, which ensures that $SYS\big|_{\mathsf{dom}(NAT)} \subseteq NAT$. Clearly, this does not allow pairs (m_1, c_2) that belong to SYS, but no to NAT.

Example 2

In Section 4.1 we mentioned that (Gunter et al., 2000) have an example that highlights the second problem we described about the system and software acceptability proposed by (Parnas and Madey, 1995), wich allows system specifications (descriptions) with outputs not physically possible. In what follows we use this example to show how our demonic approach remedies the aforementioned deficiency. In this example, the five relations of the four-variable model are

$$
\begin{aligned}
NAT &= \{(m, c) \in \mathbf{M} \times \mathbf{C} \mid \forall t \bullet m(t) < 0) \wedge (\forall t \bullet c(t) > 0)\}\,,\\
REQ &= \{(m, c) \in \mathbf{M} \times \mathbf{C} \mid \forall t \bullet c(t+3) = -m(t)\}\,,\\
IN &= \{(m, i) \in \mathbf{M} \times \mathbf{I} \mid \forall t \bullet i(t+1) = m(t)\}\,,\\
SOF &= \{(i, o) \in \mathbf{I} \times \mathbf{O} \mid \forall t \bullet o(t+1) = i(t)\}\,,\\
OUT &= \{(o, c) \in \mathbf{O} \times \mathbf{C} \mid \forall t \bullet c(t+1) = o(t)\}\,,
\end{aligned}
$$

where $t \in \mathbb{R}$ is a variable that models the time and m, c, i, and o are real-valued functions of time. The relation NAT is partial, while REQ, IN, SOF, and OUT are total functions. All five relations are internally consistent, and REQ, IN, SOF, and OUT satisfy causality, that is, their outputs do not occur before the inputs that cause them. Therefore, in principle, REQ, IN, SOF, and OUT describe implementable behaviours. However, a system implementation $IN\,;SOF\,;OUT = \{(m, c) \in \mathbf{M} \times \mathbf{C} \mid \forall t \bullet c(t+3) = m(t)\}$ produces an output

of the opposite sign than required by REQ and NAT. Such an implementation is not possible since its outputs are not allowed by the physical environment. Nevertheless, this implementation specification (description) satisfies the acceptability condition (4.3) of Parnas and Madey. The reason for this state of affairs is that the intersection between NAT and $IN \fatsemi SOF \fatsemi OUT$ is empty and thus (4.3) is trivially satisfied.

We now show how our demonic approach rejects implementations such as the one in the above example. The first step is to check if REQ is feasible with respect to NAT. We have two options, either to use Definition 4.1 or Theorem 4.2. We choose the second and show that $REQ \sqsubseteq NAT$. Since REQ is total and NAT partial, we have that $\mathsf{dom}\,(NAT) \subseteq \mathsf{dom}\,(REQ)$. Also, because REQ flips the sign of its inputs, for $m(t) < 0$ REQ will produce a $c(t) > 0$, thus $REQ\big|_{\mathsf{dom}(NAT)} \subseteq NAT$. Consequently, $REQ \sqsubseteq NAT$ and by Theorem 4.2 the system requirements REQ are feasible with respect NAT.

The second step is to check if IN, SOF, and OUT satisfy our demonic acceptability condition in Definition 4.4. Because REQ, IN, SOF, and OUT are total functions, demonic composition is the same as angelic composition and demonic refinement is the same as inclusion. Also, feasibility of REQ means that $REQ = REQ \sqcap NAT$. Thus we have that $IN \fatsemi SOF \fatsemi OUT = \{(m, c) \in \mathbf{M} \times \mathbf{C} \mid \forall t \bullet c(t+3) = m(t)\} \nsubseteq REQ$ and, accordingly, Definition 4.4 is not satisfied.

We have showed why SOF in the example by (Gunter et al., 2000) is not acceptable. A question worth asking is whether an acceptable SOF really exists. To answer this question, we use Theorem 4.5. Since $REQ = REQ \sqcap NAT$ and considering that REQ and IN are total, it is the case that $\mathsf{dom}\,(REQ \sqcap NAT) \subseteq \mathsf{dom}\,(IN)$, hence Theorem 4.5(i) holds. For Theorem 4.5(ii) to be satisfied, for $i(t+1) = m(t)$ there has to be an $o(t+2) = c(t+3) = -m(t)$ for any $t \in \mathbb{R}$. This is the case when $o(t+1) = -i(t)$ for any $t \in \mathbb{R}$. Consequently, an acceptable SOF is possible and the weakest specification of such SOF is well defined and given by Definition 4.6 and (4.15):

$$SOF_{req} = \{(i, o) \in \mathbf{I} \times \mathbf{O} \mid \forall t \bullet o(t+1) = -i(t)\}$$

4.7 Summary

The semantics of the four-variable model proposed by Parnas and Madey, which may be seen in relation algebraic terms as angelic, allows system specifications (descriptions) that are not completely consistent with the natural laws of the physical environment. To address this issue, we have redefined in the demonic calculus of relations the notion of feasibility of system requirements proposed by Parnas and Madey such that the system requirements specify for every input possible from the environment only behaviours allowed by the environment. We have also redefined in the demonic calculus of relations the system and software acceptability criterion of Parnas and Madey to not allow nonterminating or empty implementations.

In this chapter we also answered the main question of the thesis and gave a necessary and sufficient existence condition for acceptable software (Theorem 4.5). Because an acceptable SOF is a demonic mid factor of a feasible REQ through IN and OUT, the necessary and sufficient condition yields the weakest (i.e., least restrictive, most general) specification of the software requirements as the demonic mid residual of $REQ \sqcap NAT$ by IN and OUT. This constructive nature of the necessary and sufficient condition means that spending the effort for checking whether acceptable software exists is also an effort spent to derive the software requirements.

In the necessary and sufficient condition for acceptable SOF, IN and OUT are coupled. If changes are made to IN, then OUT may need to be modified as well to ensure the existence of an acceptable SOF, and vice versa. In Chapter 5 we will prove two stronger existence conditions for acceptable software that decouple IN and OUT.

Chapter 5

Separability of the Input and Output Interfaces

In this chapter we explore a practical implication of the necessary and sufficient implementability condition given in Chapter 4, Theorem 4.5. Because in this condition IN and OUT are coupled, the input and output hardware interfaces of a system that needs a relational four-variable model cannot, in general, be designed independently.

From a systems engineering perspective, we would like to be able to choose IN and OUT independently of each other, while still guaranteeing that an acceptable SOF exists. This separation of concerns would prevent changes to one interface from propagating to the other interface, an idea similar to Parnas' information hiding principle (Parnas, 1972). Then independent teams could design IN and OUT. A classic example of this is a conjecture by (Kalman, 1960) that became known as the "separation principle" or "separation theorem" for linear control systems which states that one can decompose the physical realization of a state feedback controller into two stages: an *observer* that computes a "best approximation" of the physical plant's state based upon the observations of the physical plant's outputs (i.e., monitored variables), and a *controller* that computes the control signals to the plant's inputs (i.e., controlled variables) assuming access to perfect state information from the plant. When the actual state of the plant is replaced in the controller

with the approximation computed by the observer, an optimal control results (Joseph and Tou, 1961).

In this chapter we describe an analogous result for embedded system interfacing that will allow the decoupling of the input and output hardware interfaces while still guaranteeing the ability of the software to meet the system requirements. We define the notions of observability (controllability) of the system requirements with respect to the input (output) interface and show that for a system that can be modelled by a functional four-variable model, observability and controllability allow for the separation of IN and OUT. We also show that in the general, relational four-variable model we can obtain a similar effect by strengthening either observability or controllability.

We assume that the system requirements are feasible with respect to the physical environment, that is, $REQ = REQ \intercal NAT$. Moreover, covers are used for the observability, controllability, and implementability conditions given in this chapter. Covers are more convenient when specializing the results from the relational setting to the functional setting. Using covers also results in more compact formulas.

5.1 Observability and Controllability

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

Definition 5.1. System requirements REQ are *observable* with respect to an input interface IN if there exists a demonic right factor of REQ through IN.

For system requirements REQ to be observable, Definition 5.1 requires that there exists a relation $X \subseteq \mathbf{I} \times \mathbf{C}$ such that $IN \triangleright X \sqsubseteq REQ$, as illustrated in Figure 5.1). Observability is a necessary condition for implementability since if $IN \triangleright SOF \triangleright OUT \sqsubseteq REQ$ we can take $X = SOF \triangleright OUT$. Intuitively,

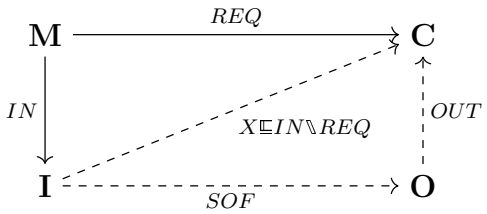


Figure 5.1: Observability in the four-variable model

observability says that in the worst case IN always retains at least as much information about the monitored variables as REQ. This point is made explicit by the following necessary and sufficient condition for observability.

Proposition 5.2. *System requirements REQ are observable with respect to an input interface IN if and only if the following conditions are both satisfied:*

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

(ii) $\mathsf{cov}\left(IN\big|_{\mathsf{dom}(REQ)}\right) \leq \mathsf{cov}\,(REQ)$.

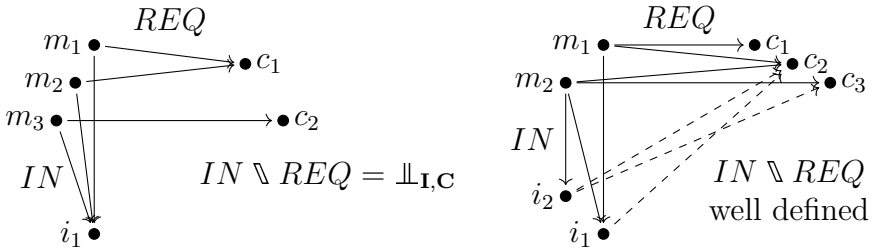
Proof. Substitute R for REQ and P for IN in Lemma 3.4. Then Proposition 5.2(i) follows trivially from Lemma 3.4(i). Also, Lemma 3.4(ii) can be rewritten as

$$\forall i \in \mathsf{ran}\left(IN\big|_{\mathsf{dom}(REQ)}\right) \bullet \exists c \in \mathbf{C} \bullet \left(IN\Big|_{\mathsf{dom}(REQ)}\right)^{\smile}(i) \subseteq REQ^{\smile}(c)\,. \tag{5.1}$$

If we take $M' = \left(IN\big|_{\mathsf{dom}(REQ)}\right)^{\smile}(i)$, then, by Definition 2.21 of covers, M' is a cell of $\mathsf{cov}\left(IN\big|_{\mathsf{dom}(REQ)}\right)$ and is indexed by $i \in \mathsf{ran}\left(IN\big|_{\mathsf{dom}(REQ)}\right)$. Similarly, $M'' = REQ^{\smile}(c)$ is a cell of $\mathsf{cov}\,(REQ)$ indexed by $c \in \mathbf{C}$. Consequently, (5.1) can be rewritten to

$$\forall M' \in \mathsf{cov}\left(IN\big|_{\mathsf{dom}(REQ)}\right) \bullet \exists M'' \in \mathsf{cov}\,(REQ) \bullet M' \subseteq M''\,, \tag{5.2}$$

which, by Definition 2.20 of refinement of covers is exactly Proposition 5.2(ii). ■



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

Figure 5.2: Examples of observability

Proposition 5.2(i) requires an input interface to "see" all the changes in monitored variables for which the requirements specify system response. Proposition 5.2(ii) requires the accuracy of the input interface to be the same or of finer granularity than what the requirements imply. For example, in Figure 5.2a, $\mathsf{cov}\left(IN\big|_{\mathsf{dom}(REQ)}\right) = \{\{m_1, m_2, m_3\}\}$ and $\mathsf{cov}\,(REQ) = \{\{m_1, m_2\}, \{m_3\}\}$. The cell $IN^{\smile}(i_1) = \{m_1, m_2, m_3\}$ in $\mathsf{cov}\left(IN\big|_{\mathsf{dom}(REQ)}\right)$ represents the accuracy with which *IN* produces i_1; in other words, the software is not able to distinguish between m_1, m_2, or m_3 when it receives the input i_1. The requirements in this example, on the other hand, require the system to make a distinction in how it treats m_3 compared to m_1 and m_2, reflected by the two distinct cells $REQ^{\smile}(c_2) = \{m_3\}$ and, respectively, $REQ^{\smile}(c_1) = \{m_1, m_2\}$ in $\mathsf{cov}\,(REQ)$. The software will not be able to make this distinction because the cell $\{m_1, m_2, m_3\}$ in $\mathsf{cov}\left(IN\big|_{\mathsf{dom}(REQ)}\right)$ is not contained in any of the cells of $\mathsf{cov}\,(REQ)$. Consequently, the accuracy of *IN* is coarser than required and *REQ* is not observable with respect to *IN*. In the example depicted in Figure 5.2b, $\mathsf{cov}\left(IN\big|_{\mathsf{dom}(REQ)}\right) = \{\{m_1, m_2\}, \{m_2\}\}$ and $\mathsf{cov}\,(REQ) = \{\{m_1\}, \{m_1, m_2\}, \{m_2\}\}$ satisfy Proposition 5.2(ii). Because $\mathsf{dom}\,(REQ) = \mathsf{dom}\,(IN)$, Proposition 5.2(i) is also satisfied, hence *REQ* is observable with respect to *IN*, ensuring that there is a way to relate the software inputs to values of controlled variables via a demonic right factor of *REQ* through *IN*. The residual $IN \setminus REQ$ is the largest, with respect to $\sqsubseteq$, such factor (i.e., the least restrictive specification of $SOF \square OUT$).

Corollary 5.3. *If system requirements REQ and input interface IN are total*

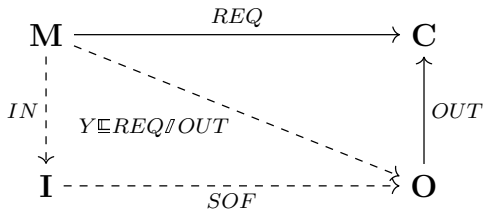


Figure 5.3: Controllability in the four-variable model

functions, then REQ is observable with respect to IN if and only if

$$\ker(IN) \leq \ker(REQ) \ .$$

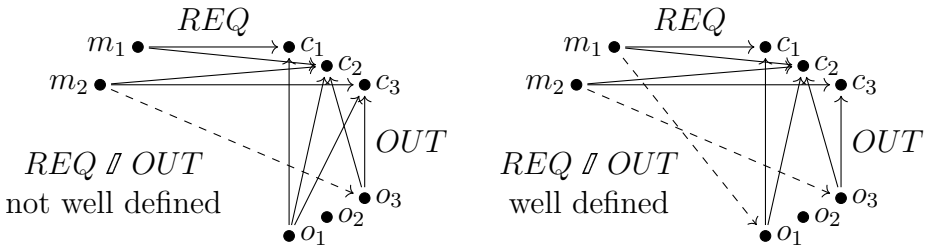
Proof. Specialize Proposition 5.2 to total functions. If REQ and IN are total functions, then $\mathsf{dom}(REQ) = \mathsf{dom}(IN)$ and Proposition 5.2(i) is trivially satisfied. Also, $IN\big|_{\mathsf{dom}(REQ)} = IN$. As explained in Section 2.4, the cover induced by a function is the same as the partition induced by the equivalence kernel of that function. As such, refinement of covers becomes refinement of kernels in the functional case and $\ker(IN) \leq \ker(REQ)$ follows from Proposition 5.2(ii). ■

Definition 5.4. System requirements REQ are *controllable* with respect to an output interface OUT if there exists a demonic left factor of REQ through OUT.

For system requirements REQ to be controllable, Definition 5.4 requires that there exists a relation $Y \subseteq \mathbf{M} \times \mathbf{O}$ such that $Y \mathbin{▫} OUT \sqsubseteq REQ$ (Figure 5.3). Similarly to observability, controllability is necessary for implementability since if $IN \mathbin{▫} SOF \mathbin{▫} OUT \sqsubseteq REQ$ we can always take $Y = IN \mathbin{▫} SOF$. The intuition for controllability is that in the worst case OUT must be at least as precise at the output (i.e., at least as deterministic) as REQ. This is made explicit by the following necessary and sufficient condition for controllability.

Proposition 5.5. *System requirements REQ are controllable with respect to an output interface OUT if and only if*

$$\forall C' \in \mathsf{cov}\left(REQ^{\smile}\right) \bullet \exists C'' \in \mathsf{cov}\left(OUT^{\smile}\right) \bullet C'' \subseteq C' \ .$$



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

Figure 5.4: Examples of controllability

Proof. If we substitute R for REQ and Q for OUT in the necessary and sufficient existence condition for a demonic left factor of Lemma 3.1 we get

$$\forall m \in \mathsf{dom}\,(REQ) \bullet \exists o \in \mathsf{dom}\,(OUT) \bullet OUT(o) \subseteq REQ(m)\,. \tag{5.3}$$

Proposition 5.5 is proved by taking $C' = REQ(m)$ as a cell of $\mathsf{cov}\left(REQ^{\smile}\right)$ indexed by $m \in \mathsf{dom}\,(REQ)$ and $C'' = OUT(o)$ a cell of $\mathsf{cov}\left(OUT^{\smile}\right)$ indexed by $o \in \mathsf{dom}\,(OUT)$. ■

For the system requirements to be controllable, Proposition 5.5 requires the output hardware to allow for the same or finer control over the controlled variables than what is implied by the requirements. The cells in the covers of $REQ^{\smile}$ or $OUT^{\smile}$ are measures of the amount of control: the smaller the cell, the more precise the control. For example, in Figure 5.4a the cell $REQ(m_1) = \{c_1, c_2\}$ in $\mathsf{cov}\left(REQ^{\smile}\right)$ does not contain any of the cells of $\mathsf{cov}\left(OUT^{\smile}\right)$. As such, OUT does not have sufficient control over the controlled variables and REQ is not controllable with respect to OUT. Figure 5.4b depicts an example where there is a way to relate the monitored variables to software outputs via a demonic left factor of REQ through OUT and, consequently, REQ is controllable. The residual $REQ \mathbin{/\!\!/} OUT$ is the largest such factor, with respect to $\sqsubseteq$ (i.e., the least restrictive specification of $IN \mathbin{\square} SOF$).

Corollary 5.6. *If system requirements REQ and output interface OUT are functions, then REQ is controllable with respect to OUT if and only if*

$$\mathsf{ran}\,(REQ) \subseteq \mathsf{ran}\,(OUT)\ .$$

Proof. By specializing the necessary and sufficient controllability condition (5.3) to functions. Because m and c are in $\mathsf{dom}\,(REQ)$ and $\mathsf{dom}\,(OUT)$, respectively, then $REQ(m)$ and $OUT(o)$ are nonempty singleton sets. Therefore (5.3) becomes

$$\forall m \in \mathsf{dom}\,(REQ) \,\boldsymbol{.}\, \exists o \in \mathsf{dom}\,(OUT) \,\boldsymbol{.}\, OUT(o) = REQ(m)\,. \tag{5.4}$$

The range of REQ satisfies the following equality:

$$\mathsf{ran}\,(REQ) = \bigcup_{m \in \mathsf{dom}(REQ)} REQ(m)\,.$$

Similarly,

$$\mathsf{ran}\,(OUT) = \bigcup_{o \in \mathsf{dom}(OUT)} OUT(o)\,.$$

Consequently, (5.4) implies that $\mathsf{ran}\,(REQ) \subseteq \mathsf{ran}\,(OUT)$. ■

Observability and controllability are dual concepts. This is apparent especially when comparing the necessary and sufficient conditions for observability as formulated in (5.2), and controllability as formulated in Proposition 5.5.

5.2 Implementability Condition Revisited

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

We first revisit the necessary and sufficient implementability condition given in Theorem 4.5 and give an equivalent condition using covers, which will be more convenient when specializing to the functional case.

Proposition 5.7. *System requirements REQ are implementable with respect to an input interface IN and an output interface OUT if and only if the following two conditions are both satisfied:*

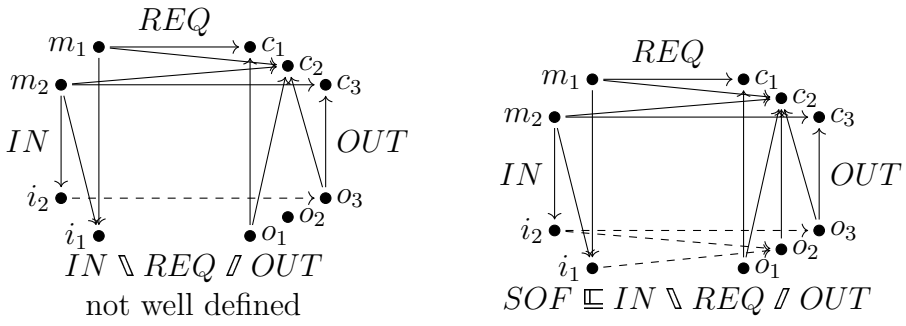
(i) $\mathsf{dom}\,(REQ) \subseteq \mathsf{dom}\,(IN)$*;*

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

Proof. Assume that system requirements are feasible with respect to the environment NAT, that is, $REQ = REQ \boxminus NAT$. Then Proposition 5.7(i) follows trivially from Theorem 4.5(i). If we take $M' = \left(IN\big|_{\mathsf{dom}(REQ)}\right)^{\smile}(i)$ for an $i \in \mathsf{ran}\left(IN\big|_{\mathsf{dom}(REQ)}\right)$ and $C' = OUT(o)$ for an $o \in \mathsf{dom}\,(OUT)$, then $M' \in \mathsf{cov}\left(IN\big|_{\mathsf{dom}(REQ)}\right)$ and $C' \in \mathsf{cov}\left(OUT^{\smile}\right)$. Thus Proposition 5.7(ii) is exactly Theorem 4.5(ii). ■

Implementability implies both observability and controllability. However, observability and controllability are not sufficient for implementability. An implication of Proposition 5.7 is that system requirements are implementable if and only if a certain balance exists between observability and controllability. A counterexample to the sufficiency of their conjunction is given in Figure 5.5a, which combines the examples from Figures 5.2b and 5.4b. In this example, REQ is both observable and controllable even though there is no acceptable software. As discussed in Section 4.5, any acceptable software implementation is a demonic refinement of the residual $IN \setminus\!\setminus REQ \mathbin{/\!\!/} OUT$, which is not well defined here. The reason for this is that i_1 cannot be connected with either o_1 or o_2 without breaking demonic refinement. For example, if we connect i_1 with o_1, then m_2 will be connected with c_1 via $IN \boxdot SOF \boxdot OUT$, something not allowed by REQ. If we extend OUT with the pair (o_2, c_2) as in Figure 5.5b, then $IN \setminus\!\setminus REQ \mathbin{/\!\!/} OUT$ becomes well defined because if we consider the cell $IN^{\smile}(i_1) = \{m_1, m_2\}$ in $\mathsf{cov}\left(IN\big|_{\mathsf{dom}(REQ)}\right)$, then there is the cell $OUT^{\smile}(o_2) = \{c_2\} = REQ(m_1) \cap REQ(m_2)$ in $\mathsf{cov}\left(OUT^{\smile}\right)$; similarly, for $IN^{\smile}(i_2) = \{m_2\}$ in $\mathsf{cov}\left(IN\big|_{\mathsf{dom}(REQ)}\right)$, there is $OUT^{\smile}(o_3) = \{c_2, c_3\} = REQ(m_2)$ in $\mathsf{cov}\left(OUT^{\smile}\right)$, hence Proposition 5.7(ii) is satisfied and an acceptable SOF exists.

Corollary 5.8. *If system requirements REQ, input interface IN and output interface OUT are total functions, then REQ is implementable with respect to IN and OUT if and only if:*



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

Figure 5.5: Implementability

(i) REQ is observable with respect to IN:

$$\ker(IN) \leq \ker(REQ) \ ;$$

(ii) and, REQ is controllable with respect to OUT:

$$\mathsf{ran}(REQ) \subseteq \mathsf{ran}(OUT) \ .$$

Proof. By specializing Proposition 5.7 to total functions.

■

In contrast to the relational case, in the functional case observability and controllability, together, are necessary and sufficient for implementability. Because observability is defined only in terms of *REQ* and *IN*, and controllability only in terms of *REQ* and *OUT*, *IN* and *OUT* are not coupled in Corollary 5.8. Thus, a practical implication of Corollary 5.8 is that the input and output interfaces of a system whose four-variable is functional can always be designed independently of each other and an acceptable software implementation is guaranteed to exist as long as *REQ* is observable and controllable with respect to *IN* and, respectively, *OUT*. This separation of concerns is not always possible for a system that needs the general, relational four-variable model. As can be seen in Proposition 5.7(ii), in the relational setting *IN* and *OUT* are coupled. The practical implications is that for the requirements to be

implementable, the input and output hardware cannot, in general, be designed independently of each other.

5.3 Separability Conditions

In this section, we present two stronger implementability conditions for the general, relational four-variable model that allow the separation of the design of the input and output hardware interfaces while still guaranteeing the implementability of the system requirements.

5.3.1 Strong Controllability

The first stronger implementability condition is obtained by strengthening controllability as follows.

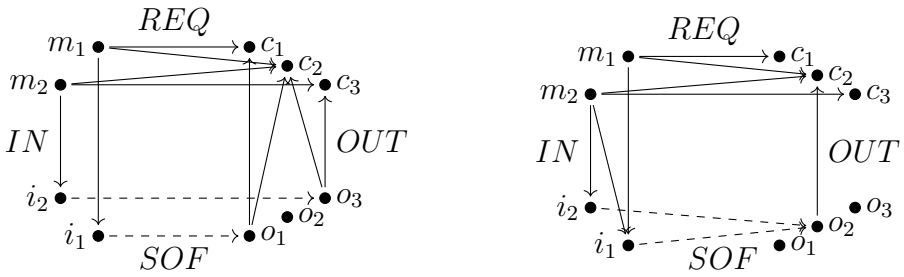
Theorem 5.9. *System requirements REQ are implementable with respect to an input interface IN and an output interface OUT if the following two conditions are both satisfied:*

(i) REQ is observable with respect to IN;

(ii) $\forall M' \in \mathsf{cov}\,(REQ) \bullet \exists C' \in \mathsf{cov}\left(OUT^{\smile}\right) \bullet C' \subseteq \bigcap_{m \in M'} REQ(m)$.

Proof. To prove the implementability of REQ we have to show that Proposition 5.7 is satisfied. Proposition 5.7(i) follows easily from Theorem 5.9(i). Also from Theorem 5.9(i), we have that for any $M' \in \mathsf{cov}\left(IN\big|_{\mathsf{dom}(REQ)}\right)$ there is a $M'' \in \mathsf{cov}\,(REQ)$ such that $M' \subseteq M''$. If we substitute M'' for M' in Theorem 5.9(ii), we get that there exists a $C' \in \mathsf{cov}\left(OUT^{\smile}\right)$ such that $C' \subseteq \bigcap_{m \in M''} REQ(m)$. Because $M' \subseteq M''$, we also have that $C' \subseteq \bigcap_{m \in M'} REQ(m)$. Thus, we have proved that for any $M' \in \mathsf{cov}\left(IN\big|_{\mathsf{dom}(REQ)}\right)$ there is a $C' \in \mathsf{cov}\left(OUT^{\smile}\right)$ such that $C' \subseteq \bigcap_{m \in M'} REQ(m)$, which is exactly Proposition 5.7(ii). ■

We call a relation REQ that satisfies Theorem 5.9(ii) *strongly controllable* with respect to OUT. An example of strongly controllable requirements is



(a) *REQ* is implementable, but not strongly controllable

(b) *REQ* is implementable, but not strongly observable

Figure 5.6: Strong observability and strong controllability are not necessary for implementability

in Figure 5.5b. Strong controllability is not necessary for implementability, as shown in Figure 5.6a. Here, the requirements are implementable and, consequently, controllable with respect to OUT, although they are not strongly controllable. As such, strong controllability reduces the choices of output devices when compared with controllability. On the other hand, strong controllability ensures that IN and OUT can be chosen independently of each other as long as they satisfy their respective constraints in Theorem 5.9.

5.3.2 Strong Observability

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

Theorem 5.10. *System requirements REQ are implementable with respect to an input interface IN and an output interface OUT if the following conditions are all satisfied:*

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

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

(iii) REQ is controllable with respect to OUT.

Proof. To prove the implementability of *REQ* we have to show that Proposition 5.7 is satisfied. Proposition 5.7(i) is exactly Theorem 5.10(i). From

Theorem 5.10(ii), we have that for any $M' \in \mathsf{cov}\left(IN\big|_{\mathsf{dom}(REQ)}\right)$ there is a $C' \in \mathsf{cov}\left(REQ^{\smile}\right)$ such that $M' \subseteq \bigcap_{c\in C'} REQ^{\smile}(c)$. If we plug C' in Theorem 5.10(iii), we obtain that there exists a $C'' \in \mathsf{cov}\left(OUT^{\smile}\right)$ such that $C'' \subseteq C'$, which implies that $M' \subseteq \bigcap_{c\in C'} REQ^{\smile}(c) \subseteq M' \subseteq \bigcap_{c\in C''} REQ^{\smile}(c)$. Consequently, $M' \subseteq \bigcap_{c\in C''} REQ^{\smile}(c)$. Since it can be shown that $M' \subseteq \bigcap_{c\in C''} REQ^{\smile}(c)$ if and only if $C'' \subseteq \bigcap_{m\in M'} REQ(m)$, we have that for any $M' \in \mathsf{cov}\left(IN\big|_{\mathsf{dom}(REQ)}\right)$ there exists a $C'' \in \mathsf{cov}\left(OUT^{\smile}\right)$ such that $C'' \subseteq \bigcap_{m\in M'} REQ(m)$, thus Proposition 5.7(ii) holds. ■

We call *REQ strongly observable* with respect to *IN* if *REQ* and *IN* satisfy Theorems 5.10(i) and 5.10(ii). An example of strongly observable requirements is in Figure 5.6a. Strong observability is not necessary for implementability (Figure 5.6b). In this example, the requirements are implementable without Theorem 5.10(ii) being satisfied. As such, strong observability restricts the acceptable choices of input hardware compared with observability, but at the same time it allows the separation of the design of *IN* and *OUT* as long as they satisfy the constraints of Theorem 5.10.

5.4 Discussion

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

- if the input interface is more difficult to design than the output interface, then it is desirable to have as many options as possible for the input devices. In such cases, Theorem 5.9 is more suitable because the implied strong controllability limits only the choices of output devices without overly restricting the input devices. If for Theorem 5.9(i) the necessary and sufficient observability condition of Proposition 5.2 is used, then Theorem 5.9 will allow the widest possible range of acceptable input hardware;

- likewise, Theorem 5.10 could be used if the output interface is more difficult to design than the input interface because the implied strong observability limits only the choices of input devices. If in Theorem 5.10(iii) the necessary and sufficient controllability condition of Proposition 5.5 is used, then Theorem 5.10 will allow the widest possible range of acceptable output devices.

If the system designers need as many acceptable options as possible for both the input and output interfaces, and separability of IN and OUT is not as important, then the necessary and sufficient implementability conditions in Proposition 5.7 should be used.

The stronger implementability conditions in Theorems 5.9 and 5.10 can be viewed as a "separation principle" for embedded systems interfacing similar to the well known separation principle for linear control systems design (Kalman, 1960). The analogy is not perfect, however. An observer in the control engineering sense would be constructed in the four-variable model as a simulation of a linear system inside SOF. The relation IN represents the input hardware that obtains the samples that would be used as input to the observer simulation. Similarly, a state feedback controller in the control engineering sense would be computed as a matrix multiplication inside SOF, the results of which would then be sent to the physical plant via the output hardware represented by OUT. Also, in control engineering observability and controllability of a plant are sufficient for separability of observers and controllers, while in the relational four-variable model either observability or controllability of REQ needs to be strengthened in order for the designs of the input and output interfaces to be separable.

5.5 Summary

In this chapter we have given two sufficient implementability conditions for the general, relational setting, that allow the design of the input and output hardware interfaces to be decoupled while still guaranteeing that an acceptable software implementation is possible.

Although the analogy is not perfect, this result is similar to the separation principle for the design of linear control systems which allows the separation of the part of the system that estimates the state of the plant based on observations of the plant's outputs, called an observer, from the part of the system that computes and sends control signals to the plant, called a controller. Likewise, we defined the concepts of observability and controllability of system requirements with respect to the input and, respectively, output interfaces. Observability of system requirements captures the capability of a system implementation to deal with every change in monitored variables for which the requirements specify system response, given a particular input interface and assuming perfect implementation for the software and output interface (diagonal **IC** in the four-variable model diagram). Controllability of system requirements denotes the capability of a system implementation to update the controlled variables only with values allowed by the requirements, given a particular output interface and assuming perfect implementation for the software and input interface (diagonal **MO** in the four-variable model diagram). In a functional four-variable model observability and controllability of system requirements are sufficient for separating the design of the input and output interfaces. In a relational four-variable model, either observability or controllability need to be strengthened to achieve such separability.

Chapter 6

Tolerances on System Requirements

For the cases when the system requirements are feasible, but an acceptable implementation does not exist, a typical engineering approach is to relax the system requirements by allowing tolerances. In this chapter we show how the necessary and sufficient implementability condition given in Chapter 4, Theorem 4.5 can be used in the derivation of tolerances on system requirements. To this end, we use an example described in (Lawford et al., 2000). In this example, the requirements of a pressure sensor trip in the shutdown system of a nuclear reactor are found to be unimplementable with respect to the chosen input and output devices. Guided by the necessary and sufficient implementability condition, we derive tolerances on the system requirements of the pressure sensor trip so that an acceptable software implementation becomes possible.

6.1 Tabular Specifications

For their readability, we will use *tabular specifications* (Parnas, 1992; Parnas, 2003), or tables for short, to describe the relations involved in the four-variable model of the pressure sensor trip system. 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 (Parnas et al., 1994; Janicki et al., 1997), (Janicki, 1995; Janicki and Khedri, 2001), (Shen et al., 1996; Zucker, 1996), (Kahl, 2003a). For the pressure sensor trip, we will use a particular type of tables, in which 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:

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

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

Tables in fact describe characteristic predicates of relations. A characteristic predicate R_{pred} of a relation $R = \{(a, b) \in A \times B \mid R_{pred}(a, b)\}$ can be seen as a function $R_{pred} : A \to B \to bool$.

A well defined tabular specification satisfies two properties: *disjointness* (i.e., the conditions in the first column do not overlap) and *completeness* (i.e., together, the conditions in the first column cover all the possible cases so that the resulting relation is total). For implementability checks we only insist on disjointness. However, the specifications of a final product must also be complete. The disjointness property ensures that the tabular specifications are internally consistent in order to avoid logical inconsistencies in the four-variable model of a system.

6.2 Example: The Pressure Sensor Trip (PST) System

In this section we study the implementability of the requirements for a pressure sensor trip (PST) subsystem of a nuclear reactor shutdown system that was described in (Lawford et al., 2000). 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 that we proved in Chapter 4, Theorem 4.5 in determining the tolerances needed on the requirements of the PST in order to be implementable given the chosen input and output devices.

6.2.1 The Four-Variable Model of the PST

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

System requirements

The computer that runs the pressure sensor trip software is connected to a pressure sensor in the plant. The software in the PST is required to make decisions as to whether a reactor shutdown procedure should be initiated or not.

Whenever the sensor value exceeds the 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. The requirements make use of a deadband region of 50 units between 2400 and 2450. For inputs in the deadband region the system is required to keep its output unchanged to prevent "tripping" and "untripping" the reactor repeatedly in a short amount of time due to sensor chatter. When the pressure is less than or equal to 2400 units, the reactor must not be tripped.

The above requirements for the PST are described formally by the fol-

lowing 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 behaviour 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 modelled by the mathematical variable *pressure* that ranges over the real numbers. The value of the previous trip state is modelled by the mathematical variable *PressTrip'* that ranges over the set $Trip = \{Tripped, NotTripped\}$. Therefore, the set **M** (Figure 1.2) of values of monitored variables 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 modelled by the controlled variable *PressTrip*, which, just as *PressTrip'*, is a member of the set *Trip*. Therefore, the set *Trip* of system outputs plays the role of the set **C** (Figure 1.2) of values of controlled variables in the four-variable model of the PST.

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.

Input interface

The input hardware interface of the pressure sensor trip computer consists of an analog-to-digital converter (ADC) for reading the monitored analog voltage produced by the sensor. The abstraction relation *R2Z* models the functionality

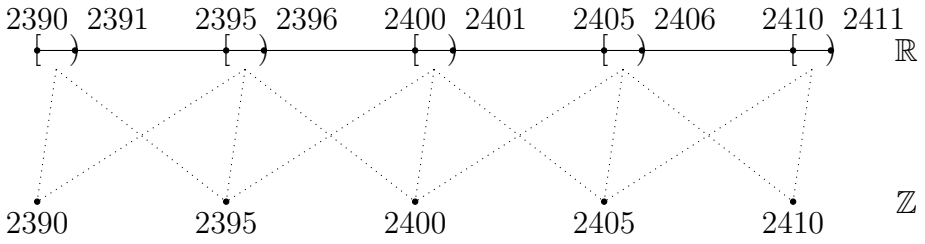


Figure 6.1: Nondeterminism introduced by the ADC

of the ADC:

$R2Z\,(pressure : \mathbb{R},\ PRES : \mathbb{Z}) : bool =$

$pressure \leq 0$	$PRES = 0$
$0 < pressure < 5000$	$\max(0, \lfloor pressure \rfloor - 5) \leq PRES \leq \lfloor pressure \rfloor + 5$
$5000 \leq pressure$	$PRES = 5000$

The variable *PRES* is a digital approximation of the actual 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) (Santina et al., 1996b; Walden, 1999; Kester, 2005). In our example, the quantization errors are modelled 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 Figure 6.1. As we will show later, this nondeterminism causes implementability issues.

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

$$Trip2Bool\,(PressTrip' : Trip,\ PREV : bool) : bool =$$

$PressTrip' = Tripped$	$PREV = true$
$PressTrip' = NotTripped$	$PREV = false$

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 (i.e, values of monitored quantities in $\mathbf{M}$), modelled as pairs of the form $(pressure, PressTrip') \in \mathbb{R} \times Trip$, to software inputs (i.e., values of software input variables in $\mathbf{I}$), modelled as pairs of the form $(PRES, PREV) \in \mathbb{Z} \times bool$:

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

Output interface

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

$$OUT\,(PTRIP : bool,\ PressTrip : Trip) : bool =$$

$PTRIP = true$	$PressTrip = Tripped$
$PTRIP = false$	$PressTrip = NotTripped$

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 is depicted in Figure 6.2.

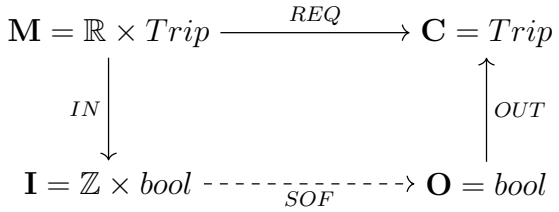


Figure 6.2: The four-variable model of the PST

6.2.2 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 requirements so they become implementable. We will use the implementability condition presented in Chapter 4, Theorem 4.5 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 4.5 to this setting, we get the following necessary and sufficient implementability condition for the pressure sensor trip system:

$$\forall (PRES, PREV) \in \mathsf{ran}\,(IN) \bullet \exists PTRIP \in \mathsf{dom}\,(OUT) \bullet \\ OUT(PTRIP) \subseteq \bigcap_{m \in IN^{\sim}(PRES,PREV)} REQ(m) \tag{6.1}$$

There are three steps in the implementability analysis we carry out for the pressure sensor trip system. First, we find all counterexamples to (6.1); this will give us the largest system input region in $\mathbf{M}$ where some, if not all, system inputs need tolerances in $\mathbf{C}$. Second, we find which of the system inputs found 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 requirements become implementable. Usually, many options 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 system input regions where tolerances are needed

The universal quantifier in (6.1) requires checking all the software inputs, which is an infinite state space. 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 (6.1) is found by taking $(PRES, PREV) = (2395, true)$. As seen in Figure 6.1, 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 Figure 6.3: 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, $\bigcap_{m \in IN^{\smile}(2395, true)} REQ(m) = \varnothing$. This constitutes a counterexample to (6.1).

If we look again at Figure 6.1, 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 is 2405. The situation at $(PRES, PREV) = (2405, true)$ is illustrated in Figure 6.4, where $\bigcap_{m \in IN^{\smile}(2395, true)} REQ(m) = \varnothing$. Consequently, $(PRES, PREV) = (2405, true)$ violates (6.1).

The cases when the previous trip state is $PressTrip' = NotTripped$ and $PREV = false$ are not problematic. The reason for this is that for

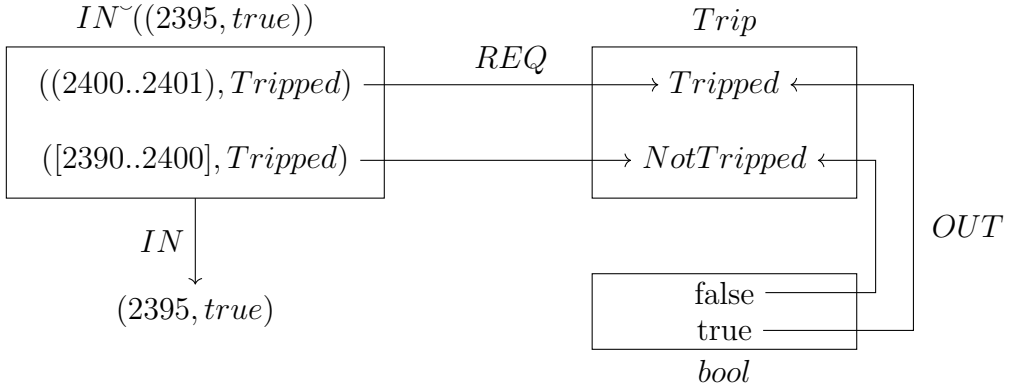


Figure 6.3: Implementability issues when $(PRES, PREV) = (2395, true)$

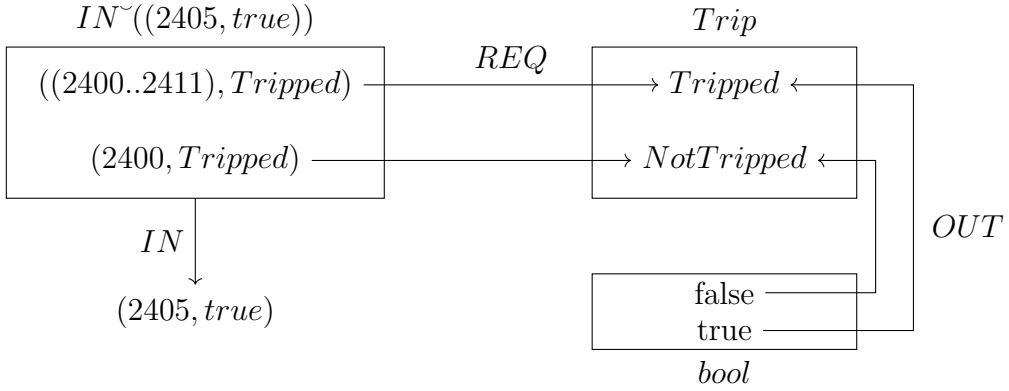


Figure 6.4: Implementability issues when $(PRES, PREV) = (2405, true)$

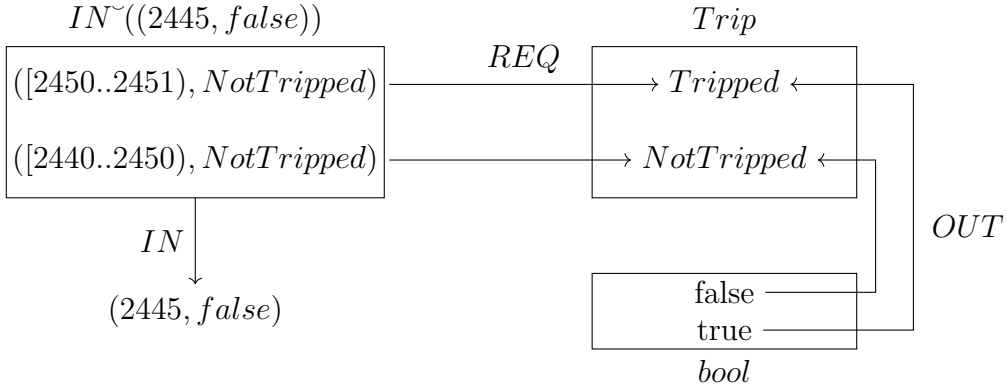


Figure 6.5: Implementability issues when $(PRES, PREV) = (2445, false)$

pressure values less than or equal to 2400 the system requirements specify that a $NotTripped$ output should be produced and that above 2400 the system output should not change. Therefore, the software inputs around the setpoint 2400 that do not satisfy (6.1) 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 (6.1) are the pairs $(PRES, PREV)$ such that $PRES$ is in the integer interval [2445..2454] and $PREV = false$. The situation at the extremes of this interval is depicted in Figures 6.5 and 6.6. 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 system input regions where there definitely are system inputs that need tolerances. Allowing tolerances outside these regions is completely unnecessary.

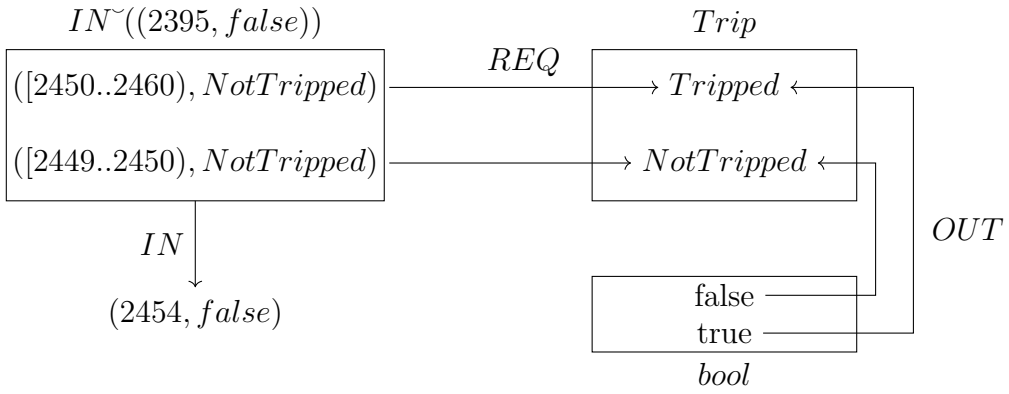


Figure 6.6: Implementability issues when $(PRES, PREV) = (2454, false)$

Step 2: Find the 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 Figures 6.3 and 6.4, 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 (6.1):

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 (6.1) (see Figures 6.5 and 6.6):

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 unnecessarily.

Step 3: Derive relaxed 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 can be used to obtain them.

Figure 6.7 depicts how the tolerances we chose to present here make an acceptable software implementation possible. For brevity, the figure illustrates only for $(PRES, PREV) = (2395, true)$ how the diagram of the four-variable model commutes.

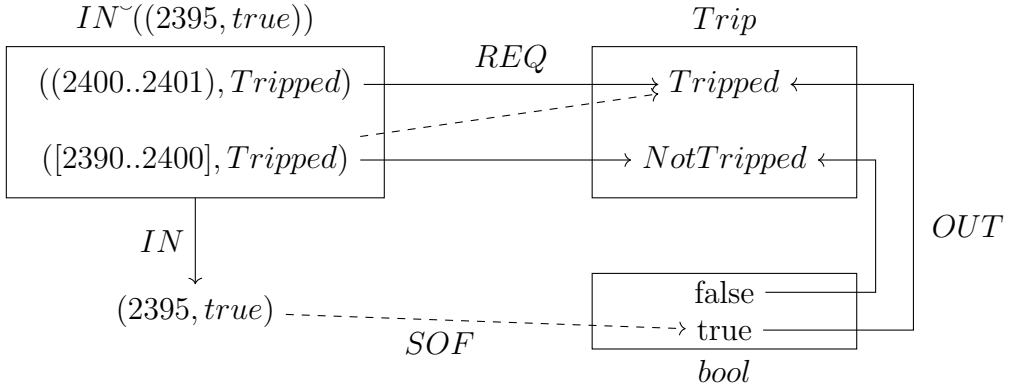


Figure 6.7: The 4-variable diagram commutes when proper tolerances are allowed on system requirements

The resulting system requirements with these 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 (6.1) has helped us to find the relaxed, implementable version *REQ'* of the original, unimplementable system requirements *REQ*. Because REQ' satisfies (6.1), 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 (6.1). 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 to ensure that the tolerances are

acceptable. We verified in Coq that *REQ'* indeed satisfies the necessary and sufficient implementability condition.

Because *REQ'* satisfies the necessary and sufficient implementability condition, the demonic mid residual $IN \setminus\!\setminus REQ \mathbin{/\!\!/} OUT$ is defined and its value (4.15) gives us the corresponding software requirements:

$$SOF_{req}((PRES, PREV) : \mathbb{Z} \times bool,\ PTRIP : bool) : bool =$$

$PRES < 2395$	$PTRIP = false$
$2395 \leq PRES < 2455$	$PTRIP = PREV$
$2455 \leq PRES$	$PTRIP = true$

6.3 Summary

In this chapter we have described how the necessary and sufficient implementability condition of Theorem 4.5 can be used in determining the tolerances needed on the requirements of a pressure sensor trip in the shutdown system of a nuclear reactor. The results of this analysis were checked with the proof assistant Coq (see Appendix B), however the analysis itself was a manual effort guided by the insight gained from the necessary and sufficient implementability condition.

The PST example was originally described in (Lawford et al., 2000), but it was incorrectly stated there that the requirements are implementable when the ADC introduces an inaccuracy of ± 5 units.

It is important to notice that usually there are many ways to relax the requirements. However, minimal changes to the requirements are desirable since it is assumed that the application domain experts had a very good reason for specifying the requirements the way they did initially. Also, once the requirements are relaxed, they will need to be revalidated.

Chapter 7

Conclusions and Future Work

We started from the belief that having a method for assessing the implementability of system requirements early in the development of a safety-critical embedded system may save development time and resources. A similar belief is shared by adepts of "lightweight formal methods", who argue that formal methods should focus on the rapid detection of faults and on providing feedback to the system designers rather than on attempting full proofs of correctness (e.g., Jackson and Wing in (Saiedian et al., 1996), or (Easterbrook and Callahan, 1998)).

Following this belief, we have developed a mathematical basis to answer the question of implementability of requirements for safety-critical embedded systems. The requirements framework that we used is the four-variable model of (Parnas and Madey, 1995). In this model, the possible behaviours of an embedded system are given by the sequential composition of the behaviours of the input devices, software, and output devices.

To be implementable, the system requirements must obey the natural laws of the physical environment in which the system is to operate, a property called *feasibility* of system requirements. Another condition for implementability is the existence of a software implementation that satisfies the constraints imposed by the system requirements and chosen hardware interfaces. Such a software implementation is called *acceptable*. We formalized the feasibility of system requirements and acceptability of software in the demonic calculus of re-

lations, strengthening the angelic definitions proposed by (Parnas and Madey, 1995). This allowed us to prove a necessary and sufficient implementability condition in the general, relational variant of the four-variable model in which inaccuracies of the input and output hardware as well as tolerances on system requirements can be modelled. The demonic setting offers guarantees of total correctness, which are more suitable for safety-critical systems than the partial correctness guarantees of an angelic setting. The demonic setting also allowed us to deal with partial specifications, which are rather the norm in early stages of system development.

Implementability is rather a theoretical property of system requirements. Implementability ensures that an acceptable software implementation exists, but does not guarantee that such software is practical to implement. If implementability is not satisfied, then no acceptable implementation is possible.

The implementability results presented in the thesis are very general. The relations *REQ*, *IN*, *OUT*, and *SOF* model input-output behaviours without internal states. Also, we did not assume any structure on the sets $\mathbf{M}$, $\mathbf{C}$, $\mathbf{I}$, and $\mathbf{O}$. 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. In our current formalization, the sets $\mathbf{M}$, $\mathbf{C}$, $\mathbf{I}$, and $\mathbf{O}$ contain all the possible values for every, respectively, monitored, controlled, input, and output variable. Time can be added explicitly to the four-variable model by treating the elements of $\mathbf{M}$, $\mathbf{C}$, $\mathbf{I}$, and $\mathbf{O}$ as functions of time (Parnas and Madey, 1995; Lawford et al., 2000; Peters, 2000). A useful research direction would be to specialize our implementability condition to include timing constraints.

To be 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 an interactive 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 software implementation 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 given by the corresponding demonic mid residual $IN \backslash\backslash (REQ \sqcap NAT) \mathbin{/\!\!/} OUT$, 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 for checking whether acceptable software is possible is also an effort spent to describe the software requirements.

A consequence of the necessary and sufficient implementability condition is that the input and output hardware interfaces of a system whose four-variable model is relational cannot be, in general, designed independently. From a systems engineering perspective, we would like this separation to be possible because it would prevent changes to one interface from propagating to the other interface. This would also allow IN and OUT to be designed by independent teams. We proved two stronger implementability conditions that allow such a separation while still guaranteeing that acceptable software is possible. This separation of concerns may increase the resilience of a system design to changes in input and output devices, but at the same time it limits the design choices for the hardware interfaces. It is an open question if the two separability conditions are too restrictive or this is the best the relational setting allows. We are more inclined towards the latter.

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 neces-

sary and sufficient implementability condition can be used in determining the tolerances needed on the requirements of a pressure sensor trip 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, in general, this may very well prove to be a hard problem, a formal characterization of common types of tolerances that are used in practice would be helpful. An example of such typical tolerances are the uniform tolerances, which allow the same deviation from the ideal behaviour for every input to the system.

The demonic factorization results presented in the thesis have applicability beyond embedded systems. They can be applied to essentially any system that can be modelled 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 behaviours at different levels of abstraction are rather 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.

For increased confidence in our results, we formalized and verified the mathematical development presented in the dissertation, as well as the implementability analysis of the pressure sensor trip system from Chapter 6, in the proof assistant Coq[1]. This may also serve as a starting point for a formal framework that offers machine support for verified system development of safety-critical systems.

[1]Coq formalization and proofs with detailed explanations are available at `www.cas.mcmaster.ca/~patcaslm/thesis/coq`

Bibliography

Back, R.-J. (1981). On correct refinement of programs. *Journal of Computer and System Sciences*, 23(1):49–68.

Back, R.-J. and von Wright, J. (1992). Combining angels, demons, and miracles in program specifications. *Theoretical Computer Science*, 100(2):365–383.

Back, R.-J. and von Wright, J. (1998). *Refinement Calculus. A Systematic Introduction*. Graduate Texts in Computer Science. Springer.

Backhouse, R. C. and van der Woude, J. (1993). Demonic operators and monotype factors. *Mathematical Structures in Computer Science*, 3(4):417–433.

Berghammer, R. and Zierer, H. (1986). Relational algebraic semantics of deterministic and nondeterministic programs. *Theoretical Computer Science*, 43:123–147.

Bharadwaj, R. and Heitmeyer, C. (2000). Developing high assurance avionics systems with the SCR requirements method. In *Proceedings of the 19th Digital Avionics Systems Conference*.

Boudriga, N., Elloumi, F., and Mili, A. (1992). On the lattice of specifications: Applications to a specification methodology. *Formal Aspects of Computing*, 4(6):544–571.

Brink, C., Kahl, W., and Schmidt, G., editors (1997). *Relational Methods in Computer Science*. Advances in Computing. Springer.

Burris, S. and Sankappanavar, H. P. (1981). A course in universal algebra. In *Graduate Texts in Mathematics*, number 78. Springer-Verlag, 2012 (millenium) edition.

Demri, S. and Orlowska, E. (1996). Logical analysis of demonic nondetermin-

istic programs. *Theoretical Computer Science*, 166(1–2):173–202.

Desharnais, J., Belkhiter, N., Sghaier, S. B. M., Tchier, F., Jaoua, A., Mili, A., and Zaguia, N. (1995). Embedding a demonic semilattice in a relation algebra. *Theoretical Computer Science*, 149(2):333–360.

Desharnais, J., Jaoua, A., Mili, F., Boudriga, N., and Mili, A. (1993). A relational division operator: The conjugate kernel. *Theoretical Computer Science*, 114(2):247–272.

Desharnais, J., Mili, A., and Nguyen, T. (1997). *Refinement and Demonic Semantics*, chapter 11, pages 166–183. In (Brink et al., 1997).

Dijkstra, E. W. (1975). Guarded commands, non-determinacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457.

Dijkstra, E. W. (1976). *A Discipline of Programming*. Prentice-Hall.

Easterbrook, S. and Callahan, J. (1998). Formal methods for verification and validation of partial specifications: A case study. *Journal of Systems and Software*, 40(3):199–210.

Faulk, S., Finneran, J., Kirby, J., Shash, S., and Sutton, J. (1994). Experience applying the CoRE method to the Lockhead C-130J software requirements. In *Ninth Annual Conference on Computer Assurance*, Gaithersburg, Maryland.

Frappier, M. (1995). *A Relational Basis for Program Construction by Parts*. PhD thesis, Computer Science Department, University of Ottawa.

Frappier, M., Mili, A., and Desharnais, J. (1996). A relational calculus for program construction by parts. *Science of Computer Programming*, 26(3):237–254.

Gunter, C. A., Gunter, E. L., Jackson, M., and Zave, P. (2000). A reference model for requirements and specifications. *IEEE Software*, 17(3):37–43.

Heimdahl, M. and Thompson, J. (2000). Specification based prototyping of control systems. In *Proceedings of the 19th IEEE Digital Avionics Systems Conference*.

Hoare, C. A. R., Hayes, I. J., Jifeng, H., Morgan, C. C., Roscoe, A. W., Sanders, J. W., Sorensen, I. H., Spivey, J. M., and Sufrin, B. A. (1987). Laws of programming. *Communications of the ACM*, 30(8):672–686.

Hoare, C. A. R. and He, J. (1985). The weakest prespecification. Technical

Monograph PRG-44, Oxford University, Computing Laboratory.

Hoare, C. A. R. and He, J. (1986). The weakest prespecification. *Fundamenta Informaticae*, 9(1):(Part I) 51–84, (Part II) 217–252.

Hoare, C. A. R. and He, J. (1987). The weakest prespecification. *Information Processing Letters*, 24(2):127–132.

Hu, X. (2008). *Proving Implementability of Timing Properties with Tolerances*. PhD thesis, Department of Computing and Software, McMaster University.

Hu, X., Lawford, M., and Wassyng, A. (2009). Formal verification of the implementability of timing requirements. In *Formal Methods for Industrial Critical Systems*, volume 5596 of *Lecture Notes in Computer Science*, pages 119–134. Springer.

Jackson, M. and Zave, P. (1993). Domain descriptions. In *Proceedings of the IEEE International Symposum on Requirements Engineering*, pages 56–64. IEEE Computer Society.

Jackson, M. and Zave, P. (1995). Deriving specifications from requirements: An example. In *Proceedings of the 17th International Conference on Software Engineering (ICSE)*, pages 15–24. ACM.

Janicki, R. (1995). Towards a formal semantics of parnas tables. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 231–240.

Janicki, R. and Khedri, R. (2001). On a formal semantics of tabular expressions. *Science of Computer Programming*, 39:189–213.

Janicki, R., Parnas, D. L., and Zucker, J. (1997). *Relational Methods in Computer Science*, chapter 12, pages 184–196. In (Brink et al., 1997).

Joseph, D. P. and Tou, T. J. (1961). On linear control theory. *Transactions of the American Institute of Electrical Engineers, Part II: Applications and Industry*, 80(4):193–196.

Kahl, W. (2003a). Compositional syntax and semantics of tables. Software Quality Research Laboratory (SQRL) 15, Department of Computing and Software, McMaster University.

Kahl, W. (2003b). Refinement and development of programs from relational specifications. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 44(3):51–93.

Kalman, R. E. (1960). Contributions to the theory of optimal control. *Boletin de la Sociedad Matematica Mexicana*, 5(2):102–119.

Kester, W., editor (2005). *The Data Conversion Handbook*. Newnes.

Knight, J. (2012). *Fundamentals of Dependable Computing for Software Engineers*. Innovations in Software Engineering and Software Development Series. Chapman & Hall/CRC.

Lawford, M., McDougall, J., Froebel, P., and Moum, G. (2000). 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)*, volume 1816 of *Lecture Notes in Computer Science*, pages 73–88. Springer.

Lempia, D. L. and Miller, S. P. (2009). Requirements engineering management handbook. Technical Report DOT/FAA/AR-08/32, U.S. Department of Transportation, Federal Aviation Administration.

Maddux, R. D. (1996). Relation-algebraic semantics. *Theoretical Computer Science*, 160(1–2):1–85.

Mili, A. (1983). A relational approach to the design of deterministic programs. *Acta Informatica*, 20(4):315–328.

Mili, A., Desharnais, J., and Mili, F. (1987). Relational heuristics for the design of deterministic programs. *Acta Informatica*, 24(3):239–276.

Miller, S. P. and Tribble, A. C. (2001). Extending the four-variable model to bridge the system-software gap. In *Proceedings of the 20th IEEE Digital Avionics Systems Conference*.

Morgan, C. (1998). *Programming from Specifications*. Prentice-Hall, 2nd edition.

Morgan, C. and Robinson, K. (1987). Specification statements and refinement. *IBM Journal of Research and Development*, 31(5):546–555.

Morris, J. M. (1987). A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058.

Parnas, D. L. (1992). Tabular representation of relations. Technical Report CRL report 260, McMaster University, Communications Research Labora-

tory.

Parnas, D. L. (2003). The tabular method for relational documentation. In *RelMiS 2001, Relational Methods in Software (a satellite event of ETAPS 2001)*, volume 44 of *Electronic Notes in Theoretical Computer Science*, pages 1–26. Elsevier Science Inc.

Parnas, D. L. and Madey, J. (1995). Functional documents for computer systems. *Science of Computer Programming*, 25(1):41–61.

Parnas, D. L., Madey, J., and Iglewski, M. (1994). Precise documentation of well-structured programs. *IEEE Transactions on Software Engineering*, 20(12):948–976.

Parnas, D. L., van Schouwen, J., and Kwan, S. P. (1990). Evaluation of safety-critical software. *Communications of the ACM*, 33(6):636–648.

Patcas, L. M., Lawford, M., and Maibaum, T. (2014a). A formal approach to implementability of safety-critical requirements for embedded systems. Submitted to Science of Computer Programming on June 7th, 2014.

Patcas, L. M., Lawford, M., and Maibaum, T. (2014b). From system requirements to software requirements in the four-variable model. In Schneider, S., Treharne, H., Margaria, T., Padberg, J., and Taentzer, G., editors, *Proceedings of the Automated Verification of Critical Systems (AVoCS) 2013*, volume 66 of *Electronic Communications of the EASST*.

Patcas, L. M., Lawford, M., and Maibaum, T. (2014c). A separation principle for embedded system interfacing. In Albert, E. and Sekerinski, E., editors, *Integrated Formal Methods (iFM)*, volume 8739 of *Lecture Notes in Computer Science*, pages 373–388. Springer.

Peters, D. K. (2000). *Deriving Real-Time Monitors from System Requirements Documentation*. PhD thesis, McMaster University.

Saiedian, H., Bowen, J. P., Butler, R. W., Dill, D. L., Glass, R. L., Gries, D., Hall, A., Hinchey, M. G., Holloway, C. M., Jackson, D., Jones, C. B., Lutz, M. J., Parnas, D. L., Rushby, J., Wing, J., and Zave, P. (1996). An invitation to formal methods. *IEEE Computer*, 29(4):16–30.

Santina, M. S., Stubberud, A. R., and Hostetter, G. H. (1996a). Discrete-time systems. In Levine, W. S., editor, *The Control Handbook*, chapter 11, pages 239–251. CRC Press and IEEE Press.

Santina, M. S., Stubberud, A. R., and Hostetter, G. H. (1996b). Quantization effects. In Levine, W. S., editor, *The Control Handbook*, chapter 15, pages 301–311. CRC Press and IEEE Press.

Schmidt, G. (2011). *Relational Mathematics*. Encyclopedia of Mathematics and its Applications. Cambridge University Press.

Schmidt, G., Hattensperger, C., and Winter, M. (1997). *Heterogeneous Relation Algebra*, chapter 3, pages 39–53. In (Brink et al., 1997).

Schmidt, G. and Ströhlein, T. (1993). *Relations and Graphs: Discrete Mathematics for Computer Scientists*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag.

Shen, H., Zucker, J., and Parnas, D. L. (1996). Table transformation tools: Why and how. In *Proceedings of the 11th Conference on Computer Assurance (COMPASS)*, pages 3–11.

Tarski, A. (1941). On the calculus of relations. *Journal of Symbolic Logic*, 6(3):73–89.

Thompson, J., Heimdahl, M., and Miller, S. P. (1999). Specification-based prototyping for embedded systems. In Nierstrasz, O. and Lemoine, M., editors, *Seventh ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, volume 1687 of *Lecture Notes in Computer Science*, pages 163–179. Springer.

Thompson, J., Whalen, M., and Heimdahl, M. (2000). Requirements capture and evaluation in NIMBUS: The light-control case study. *Journal of Universal Computer Science*, 6(7):731–757.

Van Schouwen, A. (1990). The A-7 requirements model: Re-examination for real-time systems and an application to monitoring systems. Technical Report 90-276, Queens University, Ontario, Canada.

Walden, R. H. (1999). Analog-to-digital converter survey and analysis. *IEEE Journal on Selected Areas in Communications*, 17(4):539–550.

Wassyng, A. and Lawford, M. (2003). Lessons learned from a successful implementation of formal methods in an industrial project. In Araki, K., Gnesi, S., and Mandrioli, D., editors, *FME 2003*, volume 2805 of *Lecture Notes in Computer Science*, pages 133–153. Springer.

Wassyng, A. and Lawford, M. (2006). Software tools for safety-critical soft-

ware development. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(4–5):337–354.

Wonham, W. M. (2013). Lecture notes on supervisory control of discrete-event systems. Systems Control Group, Department of Electrical & Computer Engineering, University of Toronto.

Zave, P. and Jackson, M. (1997). Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30.

Zucker, J. (1996). Transformations of normal and inverted function tables. *Formal Aspects of Computing*, 8(6):679–705.

Appendix A

Demonic Left and Right Residuals in the Literature

The demonic left and right residuals appear under various names and forms in the literature. All are equivalent, but this fact is not always obvious. Some authors define the demonic residuals using their values and then prove that they are the largest solutions, with respect to demonic refinement, of their respective inequalities. Other authors take the opposite approach and obtain the values of these residuals as theorems. In Section 2.3.2, we presented the demonic left and right residuals using the latter approach. To give the values and existence conditions for the residuals, we used notational abbreviations for the domain and range restrictions of relations, which we believe are more intuitive for use by engineers. This notation also allowed us to make the transition from abstract relations to concrete relations. In this appendix we show that the relation-algebraic expressions used in the literature for the demonic left and right residuals can be reduced to the forms we used in this dissertation.

A.1 Demonic Left Residual

The demonic left residual appears under the name of conjugate kernel in (Desharnais et al., 1993), where the following definition is given.

Definition A.1. The *conjugate kernel* of relations R and Q is $\mathsf{k}\,(R, Q) = \overline{\overline{R} \fatsemi Q^{\smile}} \cap \top \fatsemi Q^{\smile}$.

In (Desharnais et al., 1995) the conjugate kernel is called demonic left residual. Using the value of the angelic left residual as given in (2.1), the conjugate kernel can be written as

$$\mathsf{k}\,(R, Q) = R/Q \cap \top \fatsemi Q^{\smile}, \tag{A.1}$$

which is the form used in (Frappier, 1995) and (Kahl, 2003b). In this form, it is easy to see that the demonic left residual is the range restriction of R/Q to the domain of Q, or $R/Q\big|^{\mathsf{dom}(Q)}$ in the notation that we used in (2.7).

(Desharnais et al., 1993) and (Desharnais et al., 1995) proved that the domain of R being contained in the domain of $\mathsf{k}\,(R, Q)$ is a necessary and sufficient condition for the definedness of the demonic left residual. In our notation, this condition is (2.5). (Desharnais et al., 1997) and give without proof another necessary and sufficient condition:

$$R \fatsemi \top \subseteq (R/Q) \fatsemi Q \fatsemi \top .$$

(Kahl, 2003b) gives the same condition, but states it only as a sufficient condition. Using Definition 2.13 of relational vectors, this condition can be rewritten as $\mathsf{dom}\,(R) \subseteq \mathsf{dom}\,((R/Q) \fatsemi Q)$, which is (2.6).

A.2 Demonic Right Residual

We now show that the relation-algebraic formulations given in (Frappier, 1995), (Desharnais et al., 1995), and (Kahl, 2003b) for the value of the demonic right residual are equivalent to our (2.9).

(Frappier, 1995) gives the following value for the demonic right residual:

$$\begin{aligned} P \setminus\!\setminus R &= \left(\mathsf{k}\left(R^{\smile}, (R \fatsemi \top \cap P)^{\smile}\right)\right)^{\smile} \\ &= \langle \text{by (A.1) } \& \text{ conversion is an involution} \rangle \end{aligned}$$

$$
\begin{aligned}
&\Big(\Big(R^{\smile} / \left(R \fatsemi \mathbb{T} \cap P\right)^{\smile}\Big) \cap \mathbb{T} \fatsemi \left(R \fatsemi \mathbb{T} \cap P\right)\Big)^{\smile} \\
&= \langle \text{conversion distributes over intersection} \rangle \\
&\quad \Big(R^{\smile} / \left(R \fatsemi \mathbb{T} \cap P\right)^{\smile}\Big)^{\smile} \cap \left(\mathbb{T} \fatsemi \left(R \fatsemi \mathbb{T} \cap P\right)\right)^{\smile} \\
&= \Big\langle \text{(Frappier, 1995, p. 25, (20h)): in general, } (P \backslash R)^{\smile} = R^{\smile} / P^{\smile} \\
&\qquad \& \text{ conversion antidistributes over composition} \Big\rangle \\
&\quad \left(R \fatsemi \mathbb{T} \cap P\right) \backslash R \cap \left(R \fatsemi \mathbb{T} \cap P\right)^{\smile} \fatsemi \mathbb{T} \\
&= \Big(P\big|_{\mathsf{dom}(R)} \Big\backslash R\Big) \cap \Big(P\big|_{\mathsf{dom}(R)}\Big)^{\smile} \fatsemi \mathbb{T} \\
&= \Big(P\big|_{\mathsf{dom}(R)} \Big\backslash R\Big) \Big|_{\mathsf{ran}\left(P\big|_{\mathsf{dom}(R)}\right)},
\end{aligned}
$$

which is exactly (2.9).

(Kahl, 2003b) uses the totalisation operator in defining the value of the demonic right residual.

Definition A.2. The *totalisation* of a relation $R : \mathcal{A} \leftrightarrow \mathcal{B}$ is the relation $R^{\bullet} : \mathcal{A} \leftrightarrow \mathcal{B}$ such that $R^{\bullet} = R \cup \overline{R \fatsemi \mathbb{T}}$.

Then the value of the demonic right residual in (Kahl, 2003b) is:

$$
\begin{aligned}
P \mathbin{\backslash\!\!\backslash} R &\succeq (P \backslash R^{\bullet}) \cap P^{\smile} \fatsemi R \fatsemi \mathbb{T} \\
&\succeq (P \backslash R^{\bullet}) \Big|_{\mathsf{ran}\left(P\big|_{\mathsf{dom}(R)}\right)}
\end{aligned}
$$

In (Desharnais et al., 1995) the value of the demonic right residual is:

$$
\begin{aligned}
P \mathbin{\backslash\!\!\backslash} R &= \overline{P^{\smile} \fatsemi \left(\overline{R} \cap R \fatsemi \mathbb{T}\right)} \cap P^{\smile} \fatsemi R \\
&= \langle \text{double complementation } \& \text{ de Morgan's law} \rangle \\
&\quad \overline{P^{\smile} \fatsemi \overline{\overline{R} \cap \overline{R \fatsemi \mathbb{T}}}} \cap P^{\smile} \fatsemi R \\
&= \langle \text{by (2.2)} \rangle \\
&\quad \left(P \setminus \left(R \cup \overline{R \fatsemi \mathbb{T}}\right)\right) \cap P^{\smile} \fatsemi R \\
&= \langle \text{by Definition A.2} \rangle \\
&\quad (P \backslash R^{\bullet}) \cap P^{\smile} \fatsemi R
\end{aligned}
$$

$$= (P \backslash R^{\bullet}) \Big|_{\mathsf{ran}\left(P\Big|_{\mathsf{dom}(R)}\right)}$$

Proposition A.3. *For any two relations $R \subseteq A \times C$ and $P \subseteq A \times B$,*

$$P\Big|_{\mathsf{dom}(R)} \Big\backslash R = P \backslash R^{\bullet}$$

Proof. We first show that $P\Big|_{\mathsf{dom}(R)} \Big\backslash R \subseteq P \backslash R^{\bullet}$. Let $(b', c') \in P\Big|_{\mathsf{dom}(R)} \Big\backslash R$. Because $P\Big|_{\mathsf{dom}(R)} \subseteq$ and $R \subseteq R^{\bullet}$, by monotonicity of $\backslash$ it is also the case that $(b', c') \in P \backslash R^{\bullet}$.

We now show that $P \backslash R^{\bullet} \subseteq P\Big|_{\mathsf{dom}(R)} \Big\backslash R$. Let $(b', c') \in P \backslash R^{\bullet}$. There are two cases to consider. In the first case $b' \in \mathsf{ran}\left(P\Big|_{\mathsf{dom}(R)}\right)$; because $(b', c') \in P \backslash R^{\bullet}$, c' must be in the range of R, thus $(b', c') \in P\Big|_{\mathsf{dom}(R)} \Big\backslash R$. In the second case, $b' \notin \mathsf{ran}\left(P\Big|_{\mathsf{dom}(R)}\right)$ and the implication in the characteristic predicate of $P\Big|_{\mathsf{dom}(R)} \Big\backslash R$ is trivially satisfied. ■

By Proposition A.3, the values for the demonic right residual given in (Frappier, 1995), (Desharnais et al., 1995), and (Kahl, 2003b) are the same as (2.9).

Appendix B

Formalization in Coq

The mathematical development presented in the thesis has been formalized and verified in the proof assistant Coq. Here we present the organization of the source files of the formalization in Coq, as well as the connection with the material presented in the main body of the dissertation. A literate version of the complete formalization and proofs is available electronically at `www.cas.mcmaster.ca/~patcaslm/thesis/coq`.

The reader should note that some of the formal proofs in Coq do not follow the same strategy as their informal counterparts in the dissertation. This is mostly due to the peculiarities of the sequent calculus employed by Coq, which favours backwards reasoning. Presenting the proofs in this style to an audience not familiar with interactive theorem proving, and Coq in particular, would have been cumbersome.

Figure B.1 depicts the hierarchy of the Coq source files. In the sequel we present briefly that main purpose of each Coq file and how it relates to the dissertation.

Case.v Coq does not have an explicit command from moving from one brach of a proof by cases to the next. This file contains the implementation of the `Case` tactic, which was developed by the Software Foundations project[1] led by Benjamin Pierce at the University of Pennsylvania.

[1] `http://www.cis.upenn.edu/~bcpierce/sf`

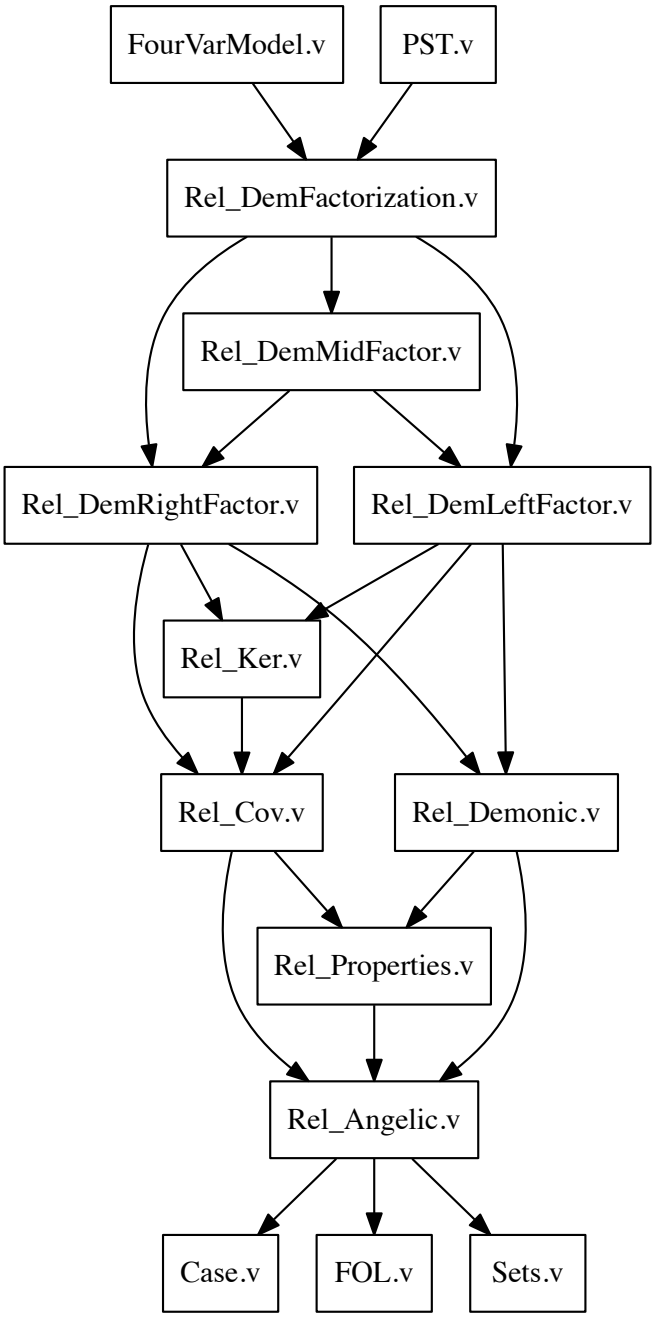


Figure B.1: Hierachy of the Coq files

FOL.v This file contains some useful theorems about First Order Logic that are not offered by the Coq Standard Library.

Sets.v A formalization of sets as predicates over a universe along with some operations on sets.

Rel_Angelic.v A formalization of the angelic calculus of concrete heterogeneous relations presented in Section 2.3.1.

Rel_Properties.v Many properties of relations, such as reflexivity, transitivity, totality, univalence etc.

Rel_Cov.v A formalization of covers induced by relations on their domains. In particular, refinement of covers is proved to be a preorder (Section 2.4).

Rel_Ker.v A formalization of equivalence kernels of functions as particular cases of covers. In particular, equivalence kernels are proved to be equivalence relations and refinement of equivalence kernels is proved to be a partial order (Section 2.4).

Rel_Demonic.v Formalizes the demonic calculus of concrete heterogeneous relations presented in Section 2.3.2.

Rel_DemLeftFactor.v The main result proved in this file is the necessary and sufficient existence condition for a demonic left factor given in Lemma 3.1, Chapter 3.

Rel_DemRightFactor.v The main result proved in this file is the necessary and sufficient existence condition for a demonic right factor given in Lemma 3.4, Chapter 3.

Rel_DemMidFactor.v The main result proved in this file is the necessary and sufficient existence condition for a demonic mid factor given in Lemma 3.7, Chapter 3.

FourVarModel.v A formalization of the demonic semantics we proposed in Chapter 4 for the four-variable model.

PST.v Verification of the implementability analysis and tolerances for the pressure sensor trip system that was presented in Chapter 6.