Chapter 12
Creating high-quality behavioural designs for software-intensive systems

Gürcan Güleşir, Pierre America, Frank Benschop, Klaas van den Berg, and Mehmet Akşit

Abstract In todays industrial practice, behavioral designs of software-intensive systems such as embedded systems are often imprecisely documented as plain text in a natural language such as English, supplemented with ad-hoc diagrams. Lack of quality in behavioral design documents causes poor communication between stake holders, up to 100 times more costly rework during testing and integration, and hard-to-maintain documents of behavioral designs. To address these problems, we present a solution that involves the usage of (a) data-flow diagrams to document the input-output relation between the actions performed by a software-intensive system, (b) control-flow diagrams to document the possible sequences of actions performed by the system, and (c) Vibes diagrams to document temporal or logical constraints on the possible sequences of actions performed by the system. The key benefit of this solution is to improve the separation of concerns within behavioral design documents; hence to improve their understandability, maintainability, and evolvability.

Gürcan Güleşir
Department of Computer Science, University of Twente, the Netherlands
e-mail: g.gulesir@cs.utwente.nl

Pierre America
Philips Research, High Tech Campus 37, 5656 AE Eindhoven, the Netherlands
e-mail: pierre.america@philips.com

Frank Benschop
Philips Healthcare, Veenpluis 4-6, 5684 PC Best, the Netherlands
e-mail: frank.benschop@philips.com

Klaas van den Berg
Department of Computer Science, University of Twente, the Netherlands
e-mail: k.g.vandenberg@cs.utwente.nl

Mehmet Akşit
Department of Computer Science, University of Twente, the Netherlands
e-mail: aksit@cs.utwente.nl

12.1 Introduction

During the life cycle of a software project, engineers continuously take decisions that influence the chances of success of the project. For example, engineers decide which features to implement, which architectural style to use, and how to structure data. During this decision-making process, earlier decisions are typically more critical, because they often limit the number of options for later decisions. From this perspective, the design phase of a software project is one of the most critical phases of the project, since some of the earliest decisions are taken during this phase.

In today's industrial practice, the design of a software-intensive system such as an embedded system is documented often as plain text in a natural language such as English, supplemented with some ad hoc and free form diagrams. Consequently, design documents are often unclear, imprecise, ambiguous, inconsistent, and hard to read and understand. This lack of quality in the design documents causes three major problems:

1. **Poor communication:** Stake holders of the design documents (mis)understand the design in different ways, which causes an excessive number of iterations during the development.
2. **Late and costly rework:** Inspections over the design documents do not reveal various design defects. These defects are discovered often during later phases, for instance testing, where it takes up to 100 times more costly rework to repair such a defect (McConnell, 2004).
3. **Poor evolvability:** Since design documents are hard to read and understand, it is often too costly to keep them up to date, as the actual design that is implemented in the source code evolves. Consequently, design documents quickly get outdated, in which case engineers loose the overview on the source code. Without such an overview, understanding and maintaining the source code becomes more difficult. Thus, software becomes less evolvable.

In this chapter, we present an intuitive summary of a solution that addresses these three problems, within the scope of the behavioural designs of software-intensive systems. This solution, which is depicted in Figure 12.1, consists of three components:

1. **Data-flow diagrams:** A visual language for documenting the input-output relation between the actions performed by a software-intensive system.
2. **Control-flow diagrams:** A visual language for documenting the possible executions of the system, in terms of the actions performed by the system.
3. **Vibes (VIsual BEhaviour Specifications) diagrams:** A visual language for documenting temporal or logical constraints on the possible executions of the system (Güleşir, 2008; Güleşir et al, 2009)[1].

The benefits of this solution are three fold: (1) A common and precise understanding of behavioural designs by their stake holders, (2) a well-defined consis-

[1] In (Güleşir, 2008; Güleşir et al, 2009), Vibes is referred as "VisuaL".

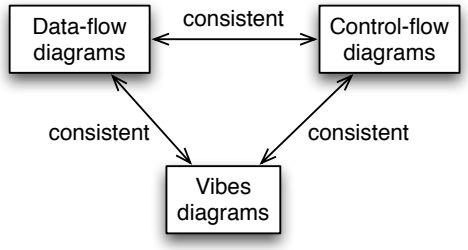

Fig. 12.1 An overview of the solution presented in this chapter.

tency relation between multiple diagrams that document a behavioural design, and (3) highly-evolvable documents of behavioural designs.

The remainder of this chapter is structured as follows: In Section 12.2, we precisely define what we mean by a behavioural design in this chapter, and then motivate the solution using an industrial example. Throughout Sections 12.3-12.6, we present data-flow diagrams, control-flow diagrams, Vibes diagrams, and discuss the consistency relation between these diagrams. Finally, in Section 12.7, we summarise this chapter.

12.2 Problems with natural language text in behavioural designs

In the context of this chapter, a **behavioural design** is a model that describes all possible executions of a system, at a certain level of abstraction. In today's industrial practice, behavioural designs are frequently documented in a natural language such as English. For example, the following text, which we refer as *txt*, is a part of a behavioural design that describes the execution of an MRI diagnostic scan:

txt During an MRI diagnostic scan, first the exam card dispatches a scan set to the scan engine. Next, the scan engine extracts the scan protocol from the scan set. And then, the scan engine writes the scan protocol in a transient storage.

Provided that the terms such as *exam card* have well-defined meanings, *txt* is precise and unambiguous, unlike most text in industrial design documents. Nevertheless, natural language texts such as *txt* are still hard to analyse and maintain on a large scale; because many types of information are mixed up in these descriptions. In *txt* for instance, six types of information are mixed up:

1. **Context:** *MRI diagnostic scan* is the context in which the behaviour is exhibited.
2. **Components:** *Exam card* and *scan engine* are the MRI system's components that exhibit the behaviour documented in *txt*.

3. **Data:** *Scan set* and *scan protocol* are two different types of data that are involved in the behaviour documented in *txt*.
4. **Actions:** *Dispatch scan set*, and *extract scan protocol* are two different types of actions that are involved in the behaviour documented in *txt*.
5. **Flow of data:** An instance of the *scan set* data flows from an execution of the *dispatch scan set* action to an execution of the *extract scan protocol* action. An instance of the *scan protocol* data flows from the execution of the *extract scan protocol* action to a transient storage.
6. **Flow of control:** The words "first" and "next" indicate that the control flows from the execution of the *dispatch scan set* action to the execution of the *extract scan protocol* action.

Mixing up multiple types of information as exemplified above, reduces the overall quality of behavioural design documents, including their understandability, maintainability, and evolvability. This problem can be seen as a specific instance of the *poor separation of concerns* problem (Dijkstra, 1982). In Sections 12.3 and 12.4, we respectively discuss the usage of data- and control-flow diagrams to document the flow of data and control *separately*; hence to improve the separation of concerns and the overall quality of behavioural design documents.

12.3 Data-flow diagrams: Input-output relation between actions

A data-flow diagram expresses the input-output relation between actions, in terms of produced and consumed instances of data. For example, the data-flow diagram in Figure 12.2 expresses the flow of data that is equivalent to the flow of data expressed by the textual description *txt* (Section 12.2). In Figure 12.2, a colour[2] represents

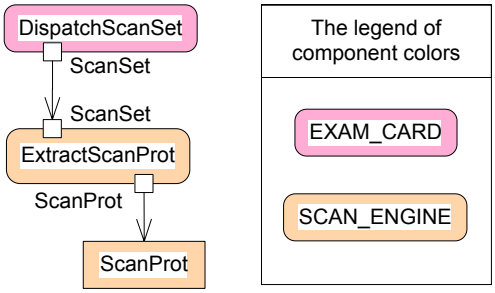

Fig. 12.2 A fragment from the complete data-flow diagram shown in Figure 12.3.

a distinct component of the MRI system, as shown in the legend of component

[2] In black and white printing the colours may not be easily distinguishable, but this distinction is not essential for understanding this chapter.

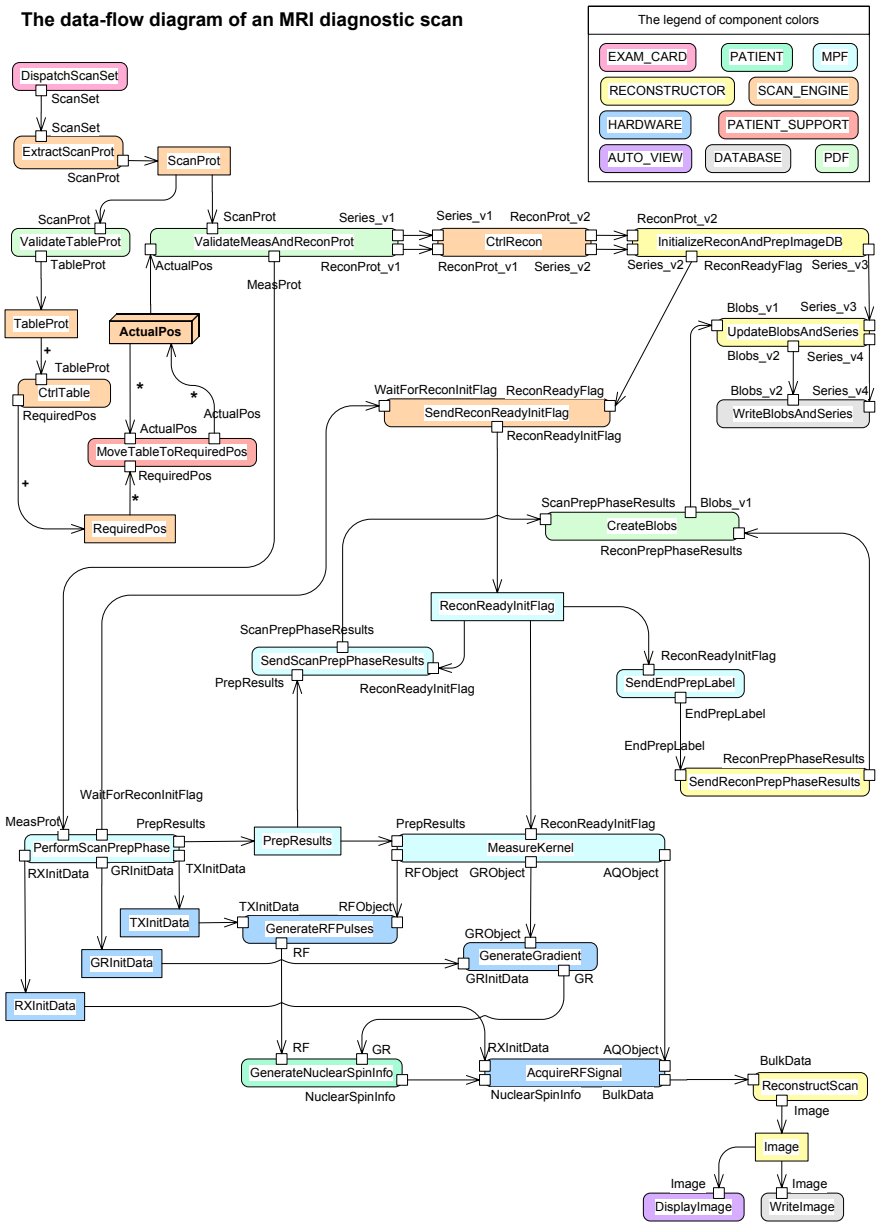

Fig. 12.3 The data-flow diagram of an MRI diagnostic scan.

colours. A coloured and rounded rectangle on the left half of Figure 12.2 represents an action performed by the component with the same colour. A small square that is attached to an action represents an input or output pin that can contain an instance of

an input or output data. A coloured rectangle represents a transient storage (similar to a variable in a programming language) that belongs to the component with the same colour. An arrow represents the flow of a data instance.

The syntax and semantics of the data-flow diagrams, which are not presented in this chapter due to the space limitation, can be seen as a specialisation of the syntax and semantics of activity diagrams in UML 2.0 (UML, http://www.uml.org/). For the sake of simplicity while interpreting and analysing the diagrams, we assume that an execution of an arrow, i.e, the flow of a data instance from the source of the arrow to the target of the arrow, is instantaneous. Accordingly, Figure 12.2 can be read as follows: "During an MRI diagnostic scan, the EXAM_CARD component performs the DispatchScanSet action, upon which this action produces an instance of the ScanSet data. When an instance of the ScanSet data is available as the output of the DispatchScanSet action, the SCAN_ENGINE component performs the ExtractScanProt action, upon which this action takes the instance of the ScanSet data as the input and produces an instance of the ScanProt data as the output. When an instance of the ScanProt data is available as the output of the ExtractScanProt action, the SCAN_ENGINE component writes this instance to the ScanProt transient storage."

One of the sizeable data-flow diagrams that were created during the Darwin project is shown in Figure 12.3. This figure contains a few additional types of shapes such as a rectangular prism and labelled arrows, which can be intuitively explained as follows: A rectangular prism, e.g., ActualPos, is a persistent storage, which always contains a data instance. Unlike a transient storage, a persistent storage can always be read without being written beforehand. A $*$-labelled arrow indicates that the flow of data occurs zero or more times during execution. A +-labelled arrow indicates that the flow of data occurs one or more times during execution. An arrow without any label indicates that the flow of data occurs exactly once during execution. These labelled arrows can be seen as new extensions to the standard activity diagrams in UML 2.0.

12.4 Control-flow diagrams: Sequences of action executions

A control-flow diagram expresses the possible executions of a system, in terms of the actions performed by the components of the system. For example, the control-flow diagram shown in Figure 12.4 expresses the flow of control that is equivalent to the flow of control expressed by the textual description *txt* (Section 12.2).

In Figure 12.4, the black dot is called the initial node, which indicates the starting point of the control flow. A coloured and rounded rectangle on the left half of Figure 12.4 represents an action. An arrow represents the flow of control from the source to the target. Accordingly, Figure 12.4 can be read as follows: "During an MRI diagnostic scan, first the EXAM_CARD component performs the DispatchScanSet action, and then the SCAN_ENGINE component performs the ExtractScanProt action."

The complete control-flow diagram of an MRI diagnostic scan is shown in Figure 12.5. This figure contains additional types of shapes such as decision and merge

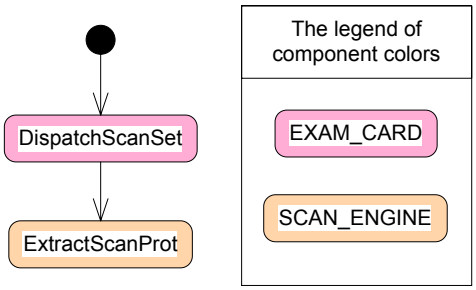

Fig. 12.4 A fragment from the complete control-flow diagram shown in Figure 12.5.

nodes, i.e., white diamonds, fork and join nodes, i.e., black bars, and flow-final nodes, i.e., crossed circles. These shapes can intuitively be explained as follows: A decision node, e.g., the white diamond after the CtrlTable action in Figure 12.5, splits an incoming flow of control into alternative outgoing flows of control. A merge node, e.g., the white diamond after the ValidateTableProt action, combines alternative incoming flows of control into a single outgoing flow of control. A fork node, e.g., the black bar after the ValidateMeasAndReconProt action, splits an incoming flow of control into parallel outgoing flows of control. A join node, e.g., the black bar after the PerformScanPrepPhase action, synchronises parallel incoming flows of control and combines them into a single outgoing flow of control. A flow-final node, e.g, the crossed circle after the WriteImage action, terminates the execution of the incoming flow of control, such that other parallel flows of control continue their execution. Each of these shapes have already been defined as a part of activity diagrams in UML 2.0.

12.5 Consistency between data- and control-flow diagrams

If there is a data-flow diagram that documents the flow of data for a specific behaviour of a system, and if there is a control-flow diagram that documents the flow of control for the same behaviour, then these two diagrams must be *consistent*. In this section, we intuitively explain this consistency relation, using the example diagrams that we introduced in the earlier sections.

Let us consider the data-flow diagram shown in Figure 12.6, which is a part of the data-flow diagram shown in Figure 12.2. The arrow shown in Figure 12.6 expresses the following two properties of an MRI diagnostic scan:

P1 During an MRI diagnostic scan, an execution of the ExtractScanProt action requires an instance of the ScanSet data as the input, and this instance is provided by an execution of the DispatchScanSet action as the output.

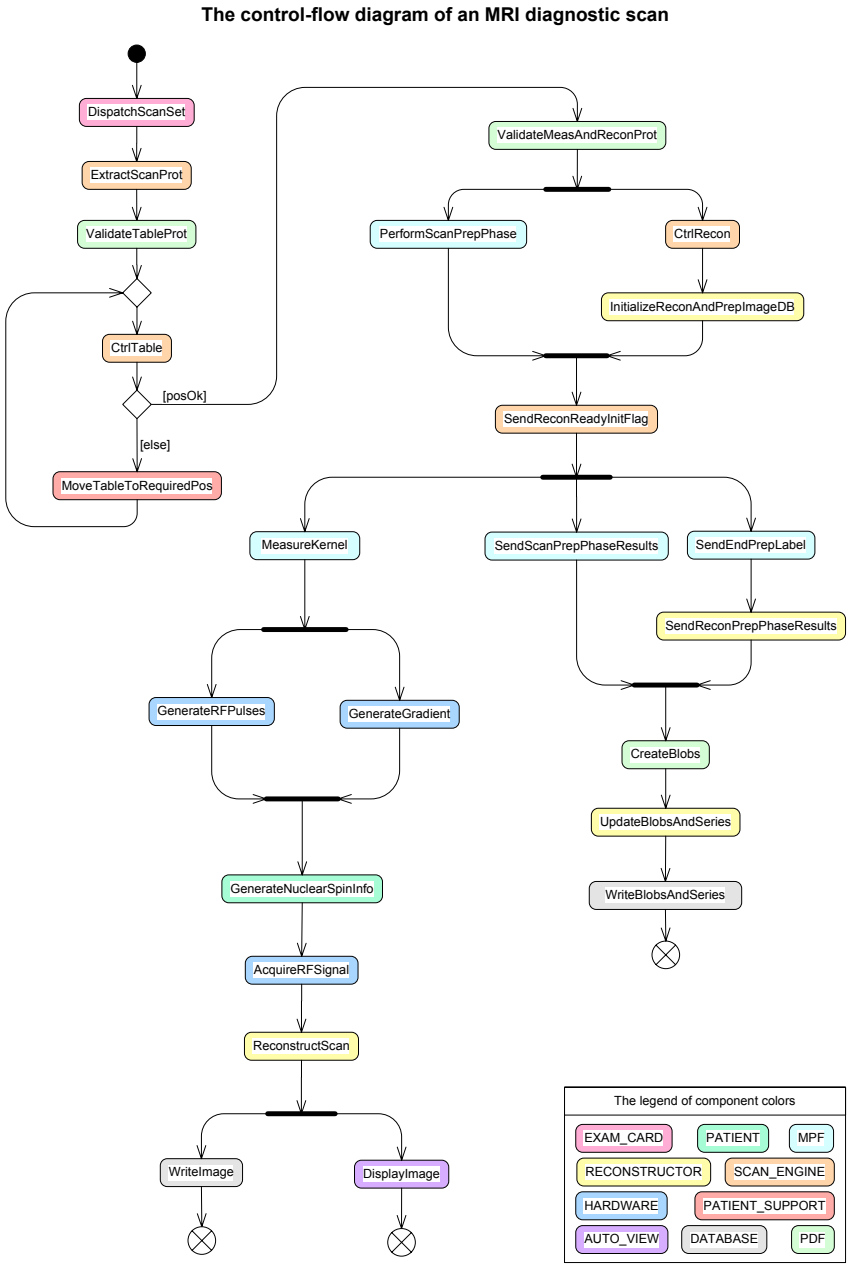

Fig. 12.5 The control-flow diagram of an MRI diagnostic scan.

P2 During an MRI diagnostic scan, exactly one instance of the ScanSet data flows from exactly one execution of the DispatchScanSet action to exactly one execution of the ExtractScanProt action.

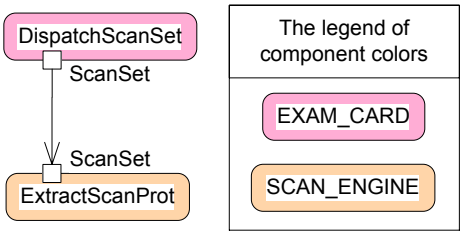

Fig. 12.6 A part of the data-flow diagram shown in Figure 12.2.

As exemplified above, each arrow in a data-flow diagram expresses certain properties of the system behaviour. Such a property also imposes certain temporal or logical restrictions on each possible sequences of action executions. We call these restrictions **behavioural constraint**s. For example, the property P1 imposes the following two behavioural constraints:

C1 During an MRI diagnostic scan, if there is at least one execution of the DispatchScanSet action, and if there is at least one execution of the ExtractScanProt action, then the first execution of the DispatchScanSet action must come before the first execution of the ExtractScanProt action.

C2 During an MRI diagnostic scan, if there are multiple executions of the ExtractScanProt action, then there must be at least one execution of the DispatchScanSet action between any two executions of the ExtractScanProt action.

The property P2 imposes the following behavioural constraint:

C3 During an MRI diagnostic scan, there must be exactly one execution of the DispatchScanSet action; there must be exactly one execution of the ExtractScanProt action; and the execution of the DispatchScanSet action must come before the execution of the ExtractScanProt action.

Now, let us sum up the discussion so far in this section: The arrows in a data-flow diagram impose behavioural constraints on the possible sequences of action executions. Since control-flow diagrams express the possible sequences of action executions, we can conclude that a data-flow diagram imposes behavioural constraints on the corresponding control-flow diagram. For example, the data-flow diagram shown in Figure 12.6 imposes the behavioural constraints C1, C2, and C3 on the control-flow diagram shown in Figure 12.4.

A control-flow diagram **satisfies** a behavioural constraint, if and only if each possible path through the control-flow diagram (i.e., each possible sequence of actions) satisfies the constraint. For example, there is only one possible path through the control-flow diagram shown in Figure 12.4, and this path is <DispatchScanSet, ExtractScanProt>. Since this path satisfies the behavioural constraint C1, we can say that the control-flow diagram shown in Figure 12.4 satisfies C1.

If a control-flow diagram satisfies all behavioural constraints expressed by the corresponding data-flow diagram, then these two diagrams are **consistent**, otherwise they are **inconsistent**. For example, the control-flow diagram shown in Figure 12.4 satisfies C1, C2, and C3. Therefore, this control-flow diagram and the data-flow diagram shown in Figure 12.6 are consistent.

12.6 Vibes diagrams: Constraints on action executions

In Section 12.5, we have seen that the flow of data is a major source of behavioural constraints. In this section, we discuss other sources of behavioural constraints, and explain how these constraints can be formally documented using Vibes diagrams.

12.6.1 Behavioural constraints that prevent race conditions

A typical source of race conditions is asynchronous access to shared data. Such race conditions can be prevented by behavioural constraints that synchronise the actions accessing the shared data. For example, let us consider the data-flow diagram shown in Figure 12.7, which is a part of the larger diagram shown in Figure 12.3. The diagram shown in Figure 12.7 indicates that the MoveTableToRequiredPos action

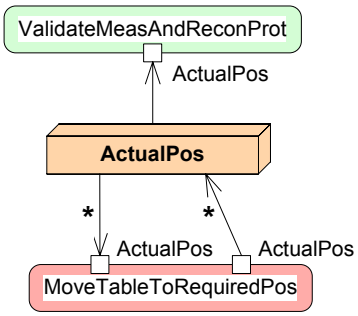

Fig. 12.7 A part of the data-flow diagram shown in Figure 12.3. Two actions asynchronously access the same data (i.e., ActualPos), which causes a race condition.

is performed zero or more times to adjust the patient table position; and upon each adjustment, this action writes the actual position of the table to the ActualPos persistent storage. In addition, the diagram indicates that the ValidateMeasAndReconProt action, when it is performed, reads the actual position of the table from the ActualPos persistent storage, and uses this information to validate the measurement and reconstruction protocols of the MRI diagnostic scan. If the actual position of the patient table changes after the validation is performed, then the quality of the resulting

diagnostic image may be low, or the image may be corrupted. Hence, the relative order in which the table movement and the validation actions are performed may have an unintended effect on the outcome of a diagnostic scan. To avoid this race condition, the following behavioural constraint is stated:

C4 During an MRI diagnostic scan, the MoveTableToRequiredPos action should not be executed after an execution of the ValidateMeasAndReconProt action.

This behavioural constraint can be formalised as the Vibes diagram shown in Figure 12.8. This diagram can be seen as a simple state-transition diagram with some wild card transitions, which have the $*$ label.

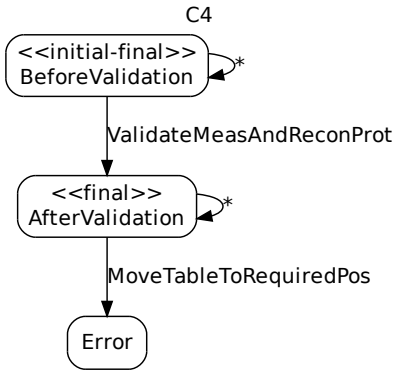

Fig. 12.8 A Vibes diagram that formally expresses the behavioural constraint C4.

To understand this diagram, we first need to view every possible execution of an MRI diagnostic scan as a sequence of actions performed by the MRI system. For example, by looking at the control-flow diagram shown in Figure 12.5, we can see that a possible execution could be represented by the following sequence of actions: $seq = <$DispatchScanSet, ExtractScanProt, ValidateTableProt, CtrlTable, MoveTableToRequiredPos, ValidateMeasAndReconProt, ... , DisplayImage$>$. A Vibes diagram either accepts or rejects such a sequence by matching each action in the sequence with a transition in the diagram. The matching starts at the initial state, which is BeforeValidation in Figure 12.8. If the matching terminates at a final state (i.e., BeforeValidation or AfterValidation in Figure 12.8), then the sequence is accepted, otherwise the sequence is rejected.

For example, let us see how the Vibes diagram shown in Figure 12.8 matches the sequence *seq* defined above. The matching starts at the initial state: BeforeValidation. Since there is no explicit transition that can match DispatchScanSet (i.e., the first action in the sequence), this action is matched by the wild card transition from BeforeValidation to BeforeValidation. Thus, a wild card transition is performed upon an action that cannot be matched by the labels of the other transitions originating from the same state.

The matching of *seq* continues as follows: Each of the ExtractScanProt, ValidateTableProt, CtrlTable, and MoveTableToRequiredPos actions are matched by the wild card transition from BeforeValidation to BeforeValidation. The ValidateMeasAndReconProt action is matched by the transition from BeforeValidation to AfterValidation. All the remaining actions are matched by the wild card transition from AfterValidation to AfterValidation. Consequently, *seq* terminates at the AfterValidation state. Since this state is a final state, the sequence *seq* is accepted by this Vibes diagram.

As one can now understand, for a given sequence to terminate at one of the final states shown in Figure 12.8, the sequence must not contain a MoveTableToRequiredPos after a ValidateMeasAndReconProt. Therefore, a given sequence of actions is accepted by this Vibes diagram, if and only if the execution of the MRI diagnostic scan represented by the sequence fulfils the behavioural constraint C4. Thus, the diagram shown in Figure 12.8 is a formal specification of C4. Güleşir (2008); Güleşir et al (2009) provide detailed information about the Vibes language.

12.6.2 Behavioural constraints derived from a domain

The domain in which a system works naturally dictates certain behavioural requirements on the system. Let us exemplify such a requirement using a more familiar system: Automated Teller Machine (ATM). An ATM is required not to dispense any cash before authenticating the user. This behavioural requirement must be fulfilled by each possible session of *any* ATM. We can therefore conclude that the domain in which an ATM operates dictates this requirement, regardless of the type or brand of the ATM.

If we assume that DispenseCash and AuthenticateUser are two of the actions that an ATM can perform, then the domain-specific behavioural requirement explained above can be translated to the following behavioural constraint:

C5 During an ATM session, the DispenseCash action should not be executed before the AuthenticateUser action is executed.

This behavioural constraint can be formalised as the Vibes diagram shown in Figure 12.9.

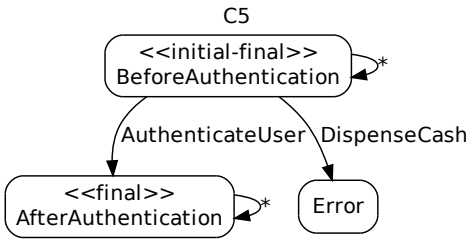

Fig. 12.9 A Vibes diagram that formally expresses the behavioural constraint C5.

12.6.3 Behavioural constraints derived from quality requirements

Let us consider the following security requirement of a generic ATM: An ATM must prevent Personal Identification Number (PIN) guessing, so that accessing an account is possible if and only if the ATM is provided with a bank card and the correct PIN associated with the bank card. Considering the majority of the existing ATMs, this security requirement is translated to the following behavioural requirement: If a user enters a wrong PIN three times during an authentication session, then the ATM must block the bank card (i.e., retain the bank card and prevent its further use). Hence, a quality requirement, which is a security requirement in this case, can be translated to a behavioural requirement, which in turn can be translated to a behavioural constraint: If we assume that RejectPIN and BlockCard are two of the actions that an ATM can perform, then the behavioural requirement explained above can be translated to the following behavioural constraint:

C6 During an ATM session, the BlockCard action must be executed immediately after the third execution of the RejectPIN action[3].

This behavioural constraint can be formalised as the Vibes diagram shown in Figure 12.10.

Note that it is not always necessary to translate a quality requirement to a behavioural requirement. For instance, the security requirement mentioned above could also be translated to a structural requirement: A PIN must consist of, say, 100 digits. Although requiring a PIN to have 100 digits makes an ATM almost 100% secure against password guessing, such a requirement makes the ATM practically unusable. Thus, the choice of how to implement or translate a quality requirement often involves a trade-off between different qualities.

Qualities such as runtime performance, response time, throughput, and energy consumption are exhibited by executions of systems. Therefore, these qualities are typical sources of behavioural requirements and constraints. For instance, the relative order in which a set of actions is performed may have an impact on the throughput of a system. In such a case, one can choose a particular order, and formulate it as a behavioural constraint, so that the desired value of throughput is achieved by the system whose all possible executions fulfil the behavioural constraint.

12.6.4 Consistency between Vibes, data- and control-flow diagrams

Vibes, data- and control-flow diagrams need to be consistent, as shown in Figure 12.1. In Section 12.5, we explained the consistency relation between data- and control-flow diagrams. In this section, we explain the remaining consistency relations.

[3] In reality, this constraint is stricter, because an ATM blocks a bank card if a wrong PIN is entered three times in a day, not necessarily during one session.

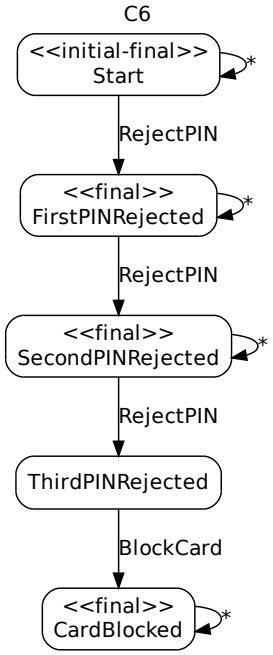

Fig. 12.10 A Vibes diagram that formally expresses the behavioural constraint C6.

A control-flow diagram is **consistent** with a Vibes diagram, if and only if each possible path through the control-flow diagram, i.e., each possible sequence of actions, are accepted by the Vibes diagram. How a Vibes diagram accepts or rejects a given sequence of actions is explained in Sec. 12.6.1.

As we explained in Section 12.5, a data-flow diagram imposes certain behavioural constraints on the flow of control. If these constraints are formally specified as Vibes diagrams, then checking the consistency between a data-flow diagram and a Vibes diagram boils down to checking the consistency between multiple Vibes diagrams. A set of Vibes diagrams are **consistent**, if and only if there is at least one sequence of actions that is accepted by each diagram in the set. If there is no such sequence, then the Vibes diagrams are **inconsistent**. Güleşir (2008) provides additional information about checking the consistency between multiple Vibes diagrams.

12.7 Summary

In todays industrial practice, behavioural designs of software-intensive systems are documented often as plain text in a natural language such as English, supplemented with some ad-hoc and free form diagrams. Consequently, the behavioural design documents are often unclear, imprecise, ambiguous, inconsistent, and hard to read

and understand. This lack of quality in behavioural design documents causes poor communication between stake holders, up to 100 times more costly rework during the software engineering life cycle, and hard-to-maintain documents of behavioural designs.

In this chapter, we presented a solution that addresses the problem stated above. This solution involves the usage of (a) data-flow diagrams to document the input-output relation between the actions performed by the components of a system, (b) control-flow diagrams to document the possible sequences of action executions, and (c) Vibes diagrams to document behavioural constraints on the possible sequences of action executions. The key benefit of this solution is to improve the separation of concerns in behavioural design documents (see Section 12.2), so that the understandability, maintainability, and evolvability of these documents are also improved.

The solution presented in this chapter is applied so far in two pilot projects and three development projects within the Philips MRI organisation. In addition, the solution is transferred to the Philips MRI organisation through four multi-site workshops conducted in Best, the Netherlands, and Cleveland, USA. In these workshops, there were approximately 40 participants, consisting of system architects, software architects, software designers, electrical engineers, and technical managers. These participants are currently using the solution in their daily practice.

Acknowledgements We would like to thank Trosky B. Callo Arias, Alexander Douglas, Piërre van de Laar, and Teade Punter for their valuable comments on earlier versions of this chapter.

References

Dijkstra EW (1982) Selected writings on computing: a personal perspective. Springer Verlag New York, Inc., New York, NY, USA

Güleşir G (2008) Evolvable behavior specifications using context-sensitive wildcards. PhD thesis, University of Twente, Enschede, DOI http://dx.doi.org/10.3990/1.9789036526333, URL http://purl.org/utwente/58767

Güleşir G, van den Berg K, Bergmans L, Akşit M (2009) Experimental evaluation of a tool for the verification and transformation of source code in event-driven systems. Empirical Software Engineering 14(6):720–777, DOI http://dx.doi.org/10.1007/s10664-009-9107-y

McConnell S (2004) Code Complete, Second Edition. Microsoft Press, Redmond, WA, USA