Improving the Quality of Requirements with Scenarios

Martin Glinz

Summary

The classical notion of requirements quality is problematic in practice. Hence the importance of some qualities, in particular completeness and unambiguity, has to be rethought. Scenarios, the description of requirements as sequences of interactions, prove to be a key concept for writing requirements specifications that are oriented towards such a modified set of qualities.

In this paper, the potential of scenarios for improving the quality of requirements is discussed. Furthermore, a concept for systematically representing both single scenarios and the structure and relationships in a set of scenarios is presented. Using an example, the positive impact of this style of representation on the quality of the requirements is demonstrated.

Martin Glinz, Professor, Institut für Informatik, Universität Zürich, Winterthurerstrasse 190, CH-8057 Zurich, Switzerland, glinz@ifi.unizh.ch, http://www.ifi.unizh.ch/~glinz

1 Introduction

The classical notion of requirements quality focuses on Adequacy[1], Unambiguity, Completeness, Consistency, Verifiability, Modifiability and Traceability (IEEE 1993). In practice however, most requirements specifications do not meet these qualities. One could argue that this is only a problem of applying the right methods and processes and that we should improve our requirements processes until they yield the desired qualities. However, a closer look reveals that it is not so simple. The qualities themselves are part of the problem.

The notion of *completeness* leads to waterfall-like process models, where a requirements specification of the complete system has to be produced and baselined prior to any design and implementation activities. However, customers do not always fully know and understand what they want. Systems and requirements evolve. So it is almost impossible to produce and freeze a complete requirements specification. *Unambiguity* requires the specification to be as formal as possible. However, in the vast majority of requirements specifications, requirements are stated informally with natural language or at best semi-formally, for example with class models or dataflow models. Thus, unambiguity is very difficult to achieve. The value of *traceability* ranges in practice from irrelevant (many in-house projects) to extremely important (safety-critical projects).

On the other hand, we also do have process- and method-related problems. In many projects, customers are unable to assess the adequacy of the requirements because the way that the requirements are represented does not match the way that customers use a system and think about it. Moreover, when customers do not fully know and understand what they want, the assessment of adequacy becomes even more difficult.

Consequently, we need both a shift in the basic paradigm of requirements quality and a proper adaptation of requirements engineering techniques in order to meet the modified set of qualities. I advocate a requirements quality model that focuses on *adequacy* as the most important quality, views *consistency*, *verifiability* and *modifiability* as next important but deliberately lives with *incompleteness* (that means with partial specifications) and with some *ambiguity*. The weight of *traceability* should be made dependent on the project in hand.

Adopting this new quality paradigm demands requirements engineering techniques that

- describe requirements such that customers can easily understand and validate them
- allow the systematic construction of partial specifications, and
- support the early detection and resolution of ambiguities.

Scenarios are a new way of representing requirements. Scenarios describe a system from a user's perspective, focusing on user-system interaction. As we will see, scenarios are a key concept for writing requirements specifications that are oriented towards the modified set of qualities discussed above.

The contribution of this paper consists of

- a general discussion of the nature of scenarios and their potential impact on requirements quality,
- a discussion of scenario representation styles and a concept for systematically structuring the scenarios in a specification.

The rest of the paper is organized as follows. In chapter two, I discuss scenarios in general, giving a

[1] In IEEE 1993, the term correctness is used for adequacy. I deliberately use adequacy instead of correctness, because correctness is a binary property (something either is correct or it is not) and implies the existence of a procedure for deciding about it. Both is typically not the case with requirements. Adequacy, however, means the quality of being able to meet a need satisfactorily, which is exactly what we want to express for requirements.

definition, listing key advantages and investigating the impact on quality. In chapter three I investigate concrete styles for scenario representation and introduce a new concept for systematically structuring scenarios. The paper concludes with a summary of achievements and open issues.

2 Scenarios

2.1 Scenarios in requirements engineering

The term "scenario" is used with several different meanings. Therefore, I start with a definition.

DEFINITION. A *scenar*io is an ordered set of interactions between partners, usually between a system and a set of actors external to the system. It may comprise a concrete sequence of interaction steps (instance scenario) or a set of possible interaction steps (type scenario).

Jacobson (1992) coined the term *use case* for type scenarios and later introduced it into UML. Hence, this term is widely used today. In Jacobson's terminology, scenarios are always on the instance level. Throughout this paper, I will use the term scenario. Where the distinction between type and instance level matters, I speak of type scenarios and instance scenarios, respectively.

Scenarios were first used in the field of human-computer interaction (Carroll 1995). Stirred by the work of Jacobson (1992), scenarios received considerable attention in requirements engineering (Firesmith 1994, Hsia et al. 1994, Carroll 1995, Glinz 1995, Weidenhaupt et al. 1998).

2.2 Key advantages of using scenarios in requirements engineering

Taking a user's viewpoint. A scenario always views a system from the viewpoint of one of its users. This is a fundamental advantage when validating the adequacy of requirements. Scenarios give the users a feel for what they will get, whereas classic techniques like lists of narrative requirements, entity-relationship diagrams or object-oriented class diagrams do not.

Partial specifications. Scenarios are a natural means for writing partial specifications. Every scenario captures a sequence of user-system interactions representing a system transaction or a system function from a user's perspective. The particular strength of scenarios lies in the fact that they provide a decomposition of a system into functions from a user's perspective and that each such function can be treated separately – a classical application of the principle of separation of concerns.

Ease of understanding. Scenarios simplify both elicitation and validation of requirements, because the notion of user-system interaction proves to be a natural way of understanding and discussing requirements both for users and requirements engineers. Scenarios inherit the comfort and ease of natural language specifications, but avoid many of the problems of a purely narrative specification.

Short feedback cycles. The combination of the ability to treat each user function separately in a scenario and the user-oriented way of representing requirements in scenarios allow short feedback cycles between users and requirements engineers.

Basis for system test. The interaction sequences captured in scenarios are also an ideal base for defining a system test. Test cases can be directly derived from the scenarios, thus improving the verifiability of requirements (Ryser and Glinz 1999).

2.3 Affected qualities of requirements

Using scenarios for the specification of requirements has a strong positive impact on the qualities of *adequacy*, *partial completeness*, *modifiability* and *verifiability* – as long as the scenarios are used properly; see chapter 3 below.

Scenarios situate requirements in the environment where a system will be used and describe them in a user-oriented way. Together with the decomposability into user functions or transactions and the ease of understanding, we have the ingredients that allow for a continuous validation of written requirements against the customers' intentions, thus yielding *adequate* specifications.

Every scenario (or group of related scenarios) represents a partial specification that is coherent from a user's perspective. Thus, *partial completeness* naturally comes with the support of partial specifications.

Using scenarios leads to a user-oriented partitioning of functionality, making it easier to deal with requirements evolution, thus improving *modifiability*.

Moreover, test cases can be derived from scenarios in a natural and straightforward way (Ryser and Glinz 1999). Thus, scenarios also are *verifiable*.

Using scenarios makes processes possible that use short cycles between writing and validating requirements and that define test cases derived from the scenarios early. Such processes, together with the user-orientation of scenarios, yield powerful capabilities for detecting and resolving ambiguities. Thus, scenarios do not lead to specifications that are a priori *unambiguous* (this is because they use natural language), but they do support processes with close feedback loops which are the natural means of detecting and resolving ambiguities in communication between humans.

Consistency is not fostered by the use of scenarios. On the contrary, viewing every scenario as a separate entity can lead to severe inconsistency problems. Moreover, not every requirement should be described by scenarios. For example, the persistent data and state-dependent behavior are easier to specify using object models and state automata.

Thus, in scenario-based approaches, explicit effort is needed to insure consistency. A systematic approach to structuring the set of scenarios as presented in the next chapter is an important step towards consistency. Consistency between scenarios and object models can be improved by systematic cross-referencing (Glinz 2000).

3 Representation of scenarios

In the last chapter, I outlined how the quality of a requirements specification can benefit from the use of scenarios. However, this benefit does not come automatically. It depends heavily on how scenarios are used and represented. A specification can be drowned in a mess of too many, badly written scenarios, making things much worse rather than making them better.

In this chapter, we investigate the impact of scenario representation on the quality of a scenario-based requirements specification and present a systematic way both for representing a single scenario and for structuring all the scenarios found in a specification.

The arguments put forward in this chapter will be illustrated by scenarios that specify a department library system. Below I give the high-level goals and constraints for this sample application.

The department library system

Goal: The system shall support a department library, where students themselves can take books from the shelves, read them, borrow / return books, and query the library catalog. Self-service terminals shall be used for borrowing and returning books.

Constraints: Users identify themselves with a personal library card. Every book has a barcode label. Additionally, there is an electronic safety device located under that barcode label which can be turned on and off by the system. At the exit of the library, a security gate sounds an alarm when a user tries to leave the library with a non-checked-out book. A user may not borrow a book if she has currently borrowed books that are overdue. A book may not be borrowed if it is reserved for another user.

3.1 The state of the art

Up to now, not too much work has been devoted to the systematic representation of scenarios, concerning both the representation of a single scenario and the structuring of a set of scenarios.

Representing a single scenario. In UML, everybody can do it the way she or he likes it (Rumbaugh, Jacobson and Booch 1999). Frequently, plain narrative text is used, in particular, when scenarios are elicited (Figure 1). Scenarios of this style are convenient for both writing and reading, but they are typically faced with serious quality problems. They are very imprecise. In particular, there is no clear sequence of actions. Exceptional cases are frequently unspecified. There is no distinction between user actions and system responses, thus making it difficult to draw a borderline between a system and its environment. For example, in the scenario of Figure 1 it is not clear whether the production of the loan slip really is the last step in the scenario or whether this feature is simply mentioned last. For the case of an illegal or unreadable card, no behavior is specified.

Type scenario: Borrow books Version: 1

When a library user wants to borrow a book, she takes it to the checkout station. There she first scans her personal library card. Then she scans the barcode label of the book. If she has no borrowed books that are overdue and the book is not reserved for another person, the systems registers the book as being borrowed by her and turns off the electronic safety device of that book. Several books can be checked out together. The checkout procedure is terminated by pressing a 'Finished' key. The system produces a loan slip for the books that have been borrowed.

Figure 1. A narrative scenario for borrowing books

Type scenario: Borrow books Version: 2
Actor: User

Normal flow
1 Scan and validate the user's library card
2 Scan book label and identify book
3 Scan label of book to be borrowed
4 Record book as borrowed, unlock safety label
5 If user wants to borrow more than one book, repeat steps 2 to 4
6 When finished, print loan slip.

Alternative flows
1.1 Card is invalid: terminate
2.1 User has overdue books: terminate
4.1 Book is reserved for another person: deny loan, continue

Figure 2. A step-by-step scenario for borrowing books

Cockburn (1997) has proposed a step-by-step description, given in natural language (Figure 2). This style is frequently used, too. Others have proposed semi-formal or formal representations, for example a sort of flow diagrams (Firesmith 1994), statecharts (Glinz 1995) or regular languages (Hsia et al. 1994).

Scenarios written in Cockburn's style exhibit a clear sequence of actions and separate normal cases from exceptional ones. However, non-linear flow of actions (alternatives, iterations) is not systematically treated and there is still no clear separation between user actions and system responses. For example, in step 2 of the scenario of Figure 2 it is not clear who (the user or the system) scans the book label and identifies the book.

Semi-formal and formal representations can be made very precise, but this advantage comes at the expense of readability and effort to write the scenarios. The strength of these approaches lies in their ability to structure a set of scenarios (see below).

Modeling structure in a set of scenarios. In UML and related approaches, there is no reasonable concept for structuring the scenarios that together form a requirements specification. The use case diagram in

UML provides two kinds of structure only: (1) it models the relationship between the scenarios/use cases and their associated actors, (2) it can express «include», «extend» and «generalize» relationships between scenarios. Neither hierarchical structure nor simple sequential composition can be expressed.

In (Glinz 1995) I have described an approach for hierarchically structuring a set of disjoint scenarios using statecharts (Harel 1987). Desharnais et al. (1998) describe a formal approach to the integration of partially overlapping sequential scenarios, which, however, requires a fully formal representation of scenarios and works under restrictive conditions only.

3.2 A new approach: systematically combining structured text and statecharts

In this section, I present a style for the representation of scenarios that combines a statechart-based structure of the set of scenarios with a structured textual representation of single scenarios.

Representation of single scenarios. Figure 3 shows a single scenario in this style. The distinctive features of this representation are

- the clear separation of the stimuli (that means the events produced by an actor) and the responses of the system,
- some simple structuring constructs (**if**, **go to step**, **terminate**) that make the flow unambiguous,
- the possibility to transform this representation into a statechart in a straightforward way[2].

Thus, we combine the readability of natural language text with the structural rigor of statecharts. Please note that (in contrast to Cockburn's notation) our normal flow can contain alternatives and iterations. This is because the normal case of a sequence of interaction steps frequently *does* include alternatives and iterations.

Structuring a set of scenarios. Statecharts provide powerful mechanisms for structuring and abstraction, which is exactly what we need for systematically organizing a set of scenarios. Structuring is required to express relations like "scenario A must be followed by scenario B" or "at this point, either scenario A or scenario B can be executed". Abstraction enables us to use scenarios both on a detailed level and on a high level and to systematically relate high-level and detailed-level scenarios with each other.

We distinguish elementary scenarios on the one hand and composite and abstract scenarios on the other. Elementary scenarios are specified textually as shown in the previous section. In composition structures, the elementary scenarios are represented as simple states. Composite and abstract scenarios are represented as statecharts. Due to the hierarchical structure of statecharts, scenarios can thus be represented on any level of abstraction. Structural relationships can easily be expressed using the features for composing statecharts. The underlying semantics of statecharts make several kinds of analyses possible and allow the symbolic execution of the scenarios.

Type scenario: Borrow books Version: 3
Actor: User

Normal flow

1	User scans her library card
	System validates the card; returns the card; displays user data; displays 'Select function' dialog
2	User selects 'Borrow' function
	System displays 'Borrow' dialog
3	User scans label of book to be borrowed
	System identifies book; records book as borrowed, unlocks safety label; displays book data
4.1	User presses 'More books' key
	System displays 'Borrow' dialog; **go to step** 3
4.2	User presses 'Finish' key
	System prints loan slip; displays 'Finished' message; **terminate**.

Alternative flows

1'	System validates the card; **if** card is invalid, system returns the card, displays 'Invalid' message; **terminate**; **endif**
2'	**if** User has overdue books, System displays 'Denied' message. **terminate**; **endif**
3'	System identifies book; **if** book is reserved for another person, system displays 'Reserved' dialog; **go to step** 4; **endif**

Figure 3. A precise textual scenario for borrowing books

In order to facilitate composition, we assume that every scenario has exactly one starting point and exactly one normal end. Hence, every statechart that represents a scenario correspondingly has one starting state and one normal exit state. We do not draw these states, we symbolize them by two bars at the top and the bottom of the statechart. An arrow ending at the top bar of a statechart represents a transition to the starting state. An arrow leaving from the bottom bar represents a transition that takes place when the statechart is in its exit state. Arrows leaving a statechart rectangle from the side represent a transition that leave the statechart whenever the trigger condition becomes true, regardless of the substate(s) that the statechart is currently in. Such transitions can be used to model exceptions. (For an example of an exception, see the timeout transitions in Figure 6.)

The notational differences between the standard statechart notation and our notation are summarized in the appendix. The notation also slightly differs from the one that was proposed in Glinz (1995) where every structural element has to be expressed as a statechart of its own. This relaxation makes reading the diagrams substantially easier.

I illustrate my approach using the library example. When viewing from a very high level, the library system has three abstract scenarios: the user uses the

[2] Every stimulus becomes an event, every response an action triggered by that event. Alternatives are modeled by states with an outgoing state transition for every alternative.

library, the librarian works in the library and the exit gate ensures that no book leaves the library that has not been checked-out. We can model this fact with the statechart of Figure 4.

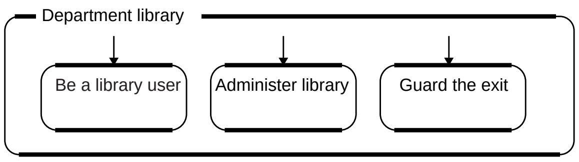

Figure 4. The three high-level scenarios of the library system

On the other hand, let us look at the scenarios for borrowing books, returning books and reserving a book that is currently on loan. These scenarios represent three alternative transactions that a user can perform in the library system. When analyzing these scenarios in more detail, we figure out that all three start with the same sequence of reading and validating the user's library card. Therefore, we factor out this authentication process into a scenario of its own and model the four scenarios as Authenticate user followed by an alternative of Borrow books, Return books or Reserve on-loan books (Figure 5). The Borrow books-scenario of Figure 3 has to be adapted accordingly by deleting steps 1 and 1'.

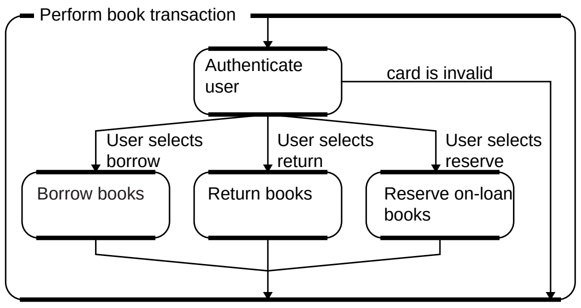

Figure 5. The structure of the transaction scenarios of the library system

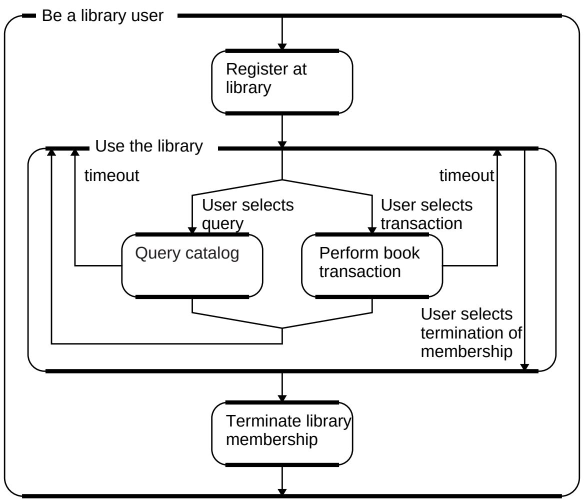

Figure 6. The structure of all user scenarios

Finally, Figure 6 models the structure of all scenarios describing interaction between users of the library and the library system. A person has to register in order to become an authorized library user. Then she or he can use the library, using the library system for performing queries of the library catalog or for executing transactions. These scenarios can be repeated arbitrarily often (see the transition leading from Query catalog and Perform book transaction back to the top of the scenario Use the library). The timeout transitions model exceptional behavior: if a user does not complete a query or a transaction, the timeout lets the system return into a defined state.

3.3 Combining scenarios with a static view of a system

As already mentioned in chapter 2, not every requirement should be described by scenarios. For example, persistent data and state-dependent behavior are easier to specify using object models and state automata. As soon as we employ more than one modeling technique for describing the requirements of a system, we have the problem of combining these models in a systematic, consistent way. A lightweight approach to consistency can be based on systematic cross-referencing between a scenario model and an object model. A more formal approach requires a model that integrates scenarios, object models and behavior models in a single integrated model (Glinz 2000).

3.4 Validation and verification capabilities

The statechart-based composition of scenarios provides us with validation and verification capabilities that go far beyond those available for a set of isolated scenarios. Exploiting the properties of statecharts, we can

- assess the adequacy of a scenario not only in isolation but also in its context,
- verify that the specification expresses required properties of a system properly, for example, mutual exclusion of scenarios that must not be executed in parallel, or the reachability of a given state of interaction,
- manage partial specifications. Statechart composition provides a framework for organizing partial specifications, keeping them consistent and showing how they fit together. It also helps to detect missing, overlapping or contradictory scenarios.

4 Conclusions

I have presented arguments for a modified paradigm of requirements quality that concentrates on adequacy, consistency, modifiability and verifiability, but requires only partial completeness and lives with some ambiguity. Achieving these qualities is more realistic than achieving the traditional ones, in particular when using incremental and evolutionary processes for software development and requirements engineering. I have shown that requirements engineering with scenarios supports the modified paradigm and potentially improves overall quality, but still needs a systematic approach to exploit this potential. Finally, I have demonstrated how a systematic, quality-oriented representation of scenarios could look, both for individual scenarios and for the relationships between a set of scenarios.

However, there are still open problems with scenario-based requirements engineering. We have neither measures nor suitable processes to determine how much formality is needed to achieve consistency and reduce ambiguity to a tolerable level. Concepts for systematic integration with other approaches are still rare and preliminary. We have no clear concepts how to integrate non-functional requirements. We still do not know how to cope systematically with different levels of requirements (business requirements vs. detailed software requirements). And we do not adequately deal with the intertwining between requirements and design where high-level design decisions produce lower-level requirements in a recursive spiral.

Nevertheless, scenario-based requirements engineering is definitely a step into the right direction. In particular, scenarios are a key enabler for an evolutionary, incremental style of requirements engineering.

Summing up, scenarios – when applied properly – do improve the quality of a requirements specification.

References

Carroll, J.M. (ed.) (1995). *Scenario-Based Design*. New York: John Wiley&Sons.

Cockburn, A. (1997). Using Goal-Based Use Cases. *Journal of Object-Oriented Programming* **10**, 7. 56-62.

Desharnais, J., Frappier, M., Khédri, R., Mili, A. (1998). Integration of Sequential Scenarios. *IEEE Transactions on Software Engineering* **24**, 9 (Sept. 1998). 695-708.

Firesmith, D.C. (1994). Modeling the Dynamic Behavior of Systems, Mechanisms and Classes with Scenarios," *Report on Object Analysis and Design*, **1**, 2. 32-36&47.

Glinz, M. (1995). An Integrated Formal Model of Scenarios Based on Statecharts. In Schäfer, W. and Botella, P. (eds.): *Software Engineering – ESEC '95. Proceedings of the 5th European Software Engineering Conference*, Sitges, Spain. LNCS 989, Berlin, etc.: Springer. 254-271.

Glinz, M. (2000). A Lightweight Approach to Consistency of Scenarios and Class Models. To appear in: *Proceedings of the Fourth IEEE International Conference on Requirements Engineering*. Schaumburg, Ill., June 2000.

Harel, D. (1987). Statecharts: A Visual Formalism for Complex Systems. *Sci. Computer Program.* **8** (1987). 231-274.

Hsia, P., Samuel, J., Gao, J., Kung, D. (1994). Formal Approach to Scenario Analysis. *IEEE Software* **11**, 2 (March 1994). 33-41.

IEEE (1993). *IEEE Recommended Practice for Software Requirements Specifications.* IEEE Std 830-1993.

Jacobson, I., Christerson, M., Jonsson, P., Övergaard, G. (1992). *Object-Oriented Software Engineering – A Use Case Driven Approach*. Reading, Mass., etc.: Addison-Wesley.

Rumbaugh, J., Jacobson, I., Booch, G. (1999). *The Unified Modeling Language Reference Manual*. Reading, Mass., etc.: Addison-Wesley.

Ryser, J., Glinz, M. (1999). A Practical Approach to Validating and Testing Software Systems Using Scenarios. *QWE'99: Third International Software Quality Week Europe*, Brussels, Nov 1999.

Weidenhaupt, K., Pohl, K., Jarke, M., Haumer, P. (1998). Scenarios in System Development: Current Practice. *IEEE Software* **15**, 2 (Mar/Apr 1998). 34-45.

Appendix: Notation for the composition of scenarios

The standard statechart notation proves to be clumsy when visualizing typical scenario structures. This is mainly because a scenario typically runs from a starting to a termination point, which requires modeling a start state and an end state in every state or statechart that represents a scenario. We therefore introduce some notational simplifications.

- We use so-called closed statecharts (Glinz 1995). These are drawn with bars at the top and the bottom of the surrounding rectangle, representing a starting state and a terminating state (Fig. A1a and b). A closed statechart is interpreted as follows.
  (1) Let A be a state representing a single scenario that is described textually in the style of Fig. 3 (Fig. A1a). When the transition leading into A is executed, the scenario becomes active and executes to termination. Then the scenario waits until the event out occurs. An occurrence of the event except signals an exception. In this case, the execution of the scenario is disrupted and the transition labeled except is taken.
  (2) Let A be a statechart representing a composite scenario (Fig. A1b). Then the execution semantics is equivalent to that of the corresponding conventional statechart given in Fig. A1c.
- We use unlabeled transitions where applicable. This is equivalent to an unconditional transition that is always enabled. It is executed immediately when its originating state is entered.
- We omit the dashed lines that usually separate parallel items in statecharts and consider any unconnected items to be parallel instead (see Fig. 4).

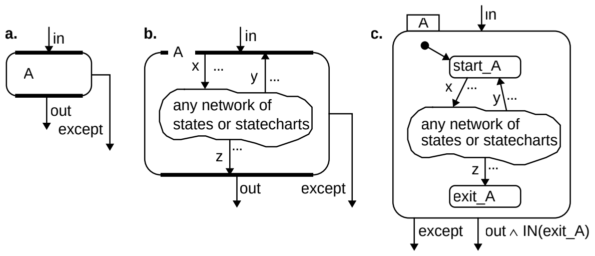

Figure A1. Interpretation of closed statecharts