Case-Based Support for Design of Dynamic System Requirements

Peter J. Funk and Dave Robertson

Edinburgh University
Department of Artificial Intelligence
80 South Bridge, Edinburgh EH1 1HN, UK
E-mail: peterf@aisb.ed.ac.uk, dr@aisb.ed.ac.uk

Published in:

Funk, P.J., Robertson D., Case-Based Support for the Design of Dynamic System Requirements. In: Advances in Case-Based Reasoning, Selected Papers, Keane M., Haton J.P., Manago M. (eds.), Springer-Verlag (ISBN 3-540-60364-6), pp. 211-225, 1995.

This paper is an extended and revised version of the paper published by the authors at the *Second European Workshop on Case-Based Reasoning, Proceedings*, France, pp. 293-301, 1994.

Case-Based Support for the Design of Dynamic System Requirements*

Peter J. Funk and Dave Robertson

Edinburgh University, Department of Artificial Intelligence
80 South Bridge, Edinburgh EH1 1HN, UK
E-mail: {peterf | dr}@aisb.ed.ac.uk

Abstract. Using formal specifications based on varieties of mathematical logic is becoming common in the process of designing and implementing software. Formal methods are usually intended to include all important details of the final system in the specification with the aim of proving that it possesses certain mathematical properties. In large, complex systems, this task requires sophisticated theorem proving, which can be difficult and complicated. Telecommunication systems are large and complex, making detailed formal specification impractical with current technology. However roughly formal "sketches" of the behaviours these services provide can be produced, and these can be very helpful in locating which service might be relevant to a given problem. Our case-based approach uses coarse-grained requirements specification sketches to outline the basic behaviour of the system's functional modules (called services), thereby allowing us to identify, reuse and adapt requirements (from cases stored in a library) to construct new cases. By using cases that have already been tested, integrated and implemented, less effort is needed to produce requirements specifications on a large scale. Using a hypothetical telecommunication system as our example, we shall show how comparatively simple logic can be used to capture coarse-grained behaviour and how a case-based approach benefits from this. The input from the examples is used both to identify the cases whose behaviour corresponds most closely to the designer's intentions and to adapt and finally verify the proposed solution against the examples.

1. Definition of Problem

One of the main problems facing designers is adding changes and modifications to an existing system in order to meet new demands. Because telecommunications systems are long-term investments, existing systems are constantly being extended to meet new demands from customers. As a rule, most code in a telecommunications system is successively replaced over a fifteen-year period (and the appropriate hardware updates made), so that the system can meet all its new requirements.

The purchase of new telecommunication services is a matter of importance to both customers and suppliers, therefore today new services are very carefully defined, albeit in informal documents which are often contractually binding. Our CABS system (CAse-Based requirements Specification [10, 11]), takes behavioural examples which

* This research was supported by the Marcus Wallenberg Foundation for Scientific Research and Education and EUA Telecommunication Systems Laboratories, Sweden.

define a service's desired behaviour (see figure 1) as input and produces transition rules (called partial rules) that cover the same behaviour as the input examples. These partial rules are then used in a matching process to identify similar, previously formalised services and transition rules, which have already been tested and integrated

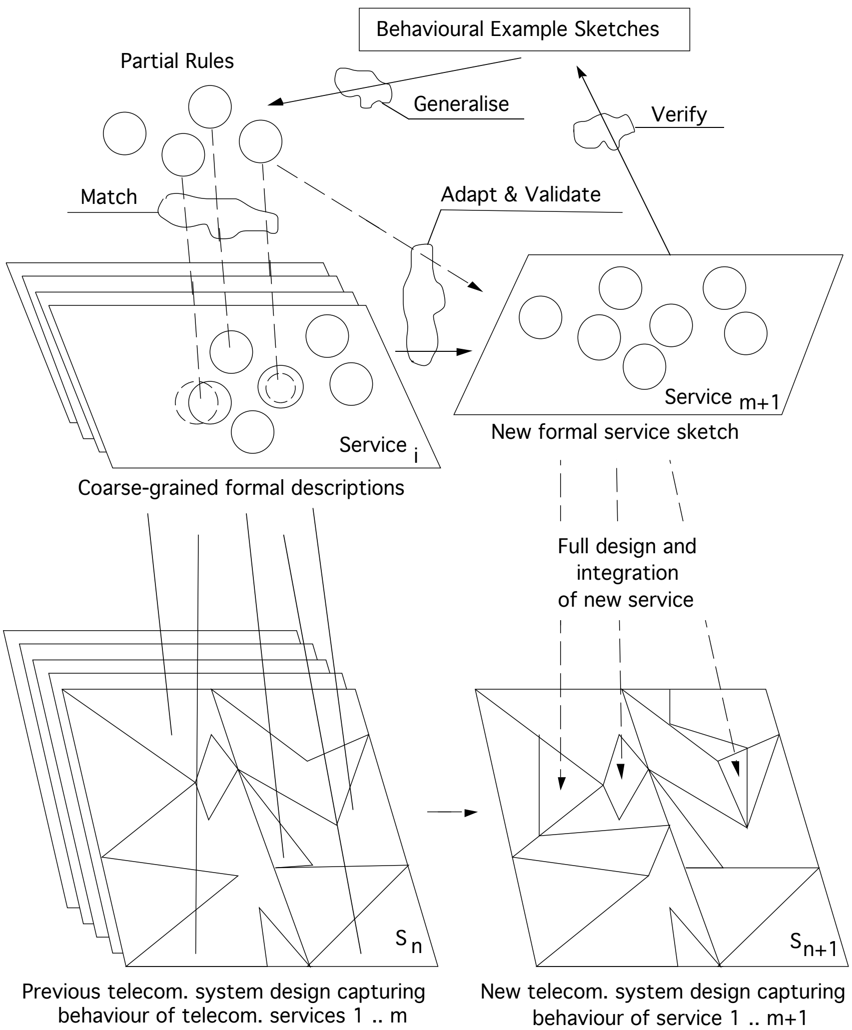

Fig. 1. Behavioural example, services and the full design of the system.

with other services. By reusing them, we should reduce the time needed to develop, test and integrate the new service requirements with the other services it has to interact with. This allows us at an early stage to provide the customer and supplier with the ability to explore the dynamic behaviour of the new service (by simulation of the formalised requirements), before any time and effort has been spent on design and implementation. It is very beneficial to clarify and correct any disagreement on functionality at this stage.

Each sheet in the bottom left corner of the diagram represents a collection of previously designed and integrated services, composed of very complex configurations of system components. The new service requirements (of which the formalised requirements is a small but important part) has also to be designed and integrated. Formalised requirements can be used in a variety of ways to enhance the traditional software development process [4], e.g., as a reference by which to guide design, to generate test cases [24], and to map onto design components [22], etc. If we have access to the relation between all previously designed and implemented parts, and their originating coarse-grained service requirements, we may assist the designers in choosing parts for reuse, by pointing out where modifications have to be made, when producing a design of the new service (thus taking us to our end point in the bottom right corner of figure 1). We do not address the task of producing a final design.

The main objective of this paper is to give an overview of our approach, which involves combining case-based reasoning with formal methods in order to benefit from the reuse of previously formulated requirements in the design of large systems. Section 2 briefly describes CABS relations to formal methods. Section 3 examines CABS in its context of case-based reasoning. Section 4 gives a brief overview and some examples of the logic used for representing cases in the case library. Section 5 gives an example of input to CABS, and explains how transition rules are generated from it. Section 6 examines how input cases are matched to cases stored in the case library. This section also offers illustrations of the set theoretical approach as well as the pseudo code for the matching algorithm. Section 7 gives a brief account of how specifications are adapted and tested. Section 8 explores some related work. Finally, section 9 summarises the research.

2. Requirements Specifications

Much effort has been made over many years to bring formal methods into use in industry. The fact that they are nevertheless not widely used may indicate that they are not yet mature, or that they are misunderstood by industry, or that industry has difficulty integrating them into current software development processes [15]. Although individual elements of a reactive system's behaviour may be amenable to representation and verification using formal methods, scaling up this approach to the specification of large complex systems appears to be difficult.

If we are to specify the complete behaviour of a large reactive system in detail in a single formalism, we may end up needing sophisticated logics and sets of axioms that can handle concurrency, time constraints, indeterminism, asynchronism, statistics, etc. The resulting complexity of proving theorems and simulating dynamic behaviour can be difficult to handle. In addition, many of the formalisms used for complex specifications are not "executable", and therefore do not allow developers to explore the dynamics of the specifications.

However, if we simply wish to outline the original requirements, as opposed to providing a complete formal specification (including error handling, odd cases, unusual interactions, etc. [28]), it is usually sufficient to consider a simplified view. We call such a view a requirements specification, since these are the original requirements, not a complete specification. We note in passing that these are not the only possible requirements which one might collect for such a system – they are merely a particular type of functional requirement.

3. Case-Based Reasoning for Requirements Capture

CABS is closely structured accordingly to the four REs (Retrieve, Reuse, Revise, Restore, [1]) in the case-based reasoning cycle (see figure 2). CABS uses a simple predicate logic to represent, in the case library, only the coarse-grained behaviour of functional elements that have already been designed and implemented. This logic is able to represent stimuli, facts, responses and simple transition rules (as shown in section 4). A case in the case library is a set of transition rules (a service). The logic

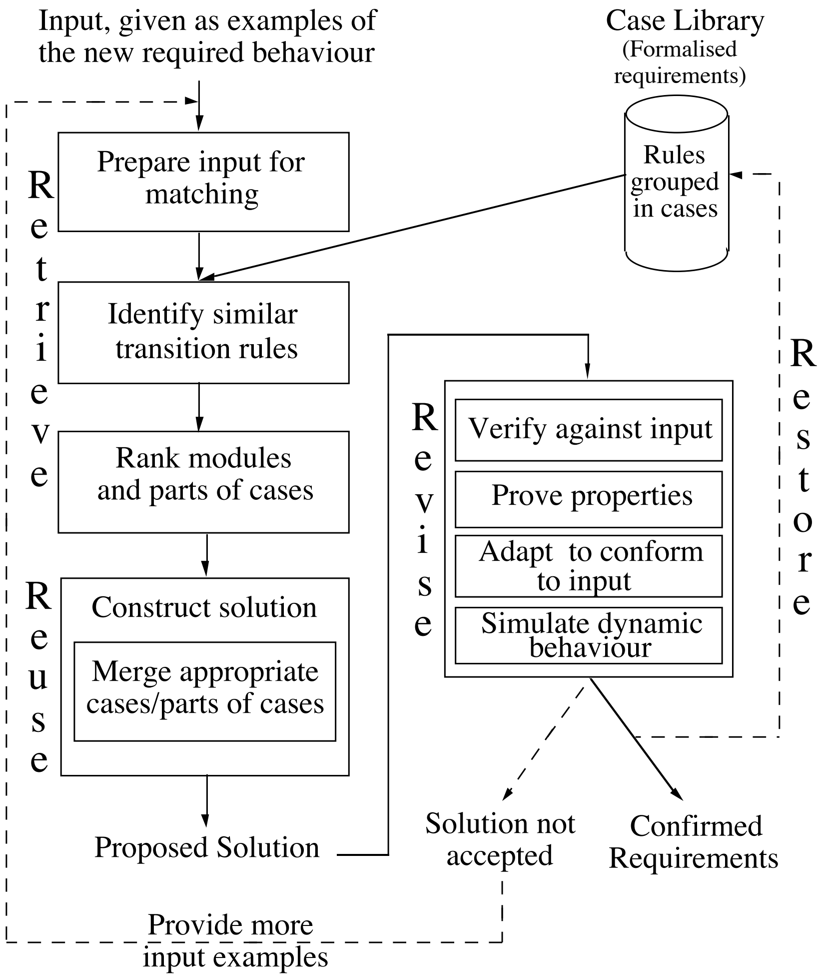

Fig. 2. Overview over the CABS system

used to represent the dynamic behaviour of cases gives us access to the coarse-grained, dynamic behaviour of each case – which provides the basis for choosing fully specified indexes [19] and for testing new cases.

The designer provides input to the system in the form of examples of the new required behaviour and the first task (upper left box in figure 2) is to prepare the input for the matching. CABS translates the input examples to a set of transition rules, which are under-specified since they do not give all details and only describe parts of the behaviour and hence the produced transition rules are called partial rules (see section 5 and 6).

The partial rules enable us to determine the similarity between the behaviour of a case in the case library and the behaviour of the new case outlined by the designer (and to indicate where the differences lie). The system identifies the cases that exhibit behaviour most similar to the new behaviour and uses them to construct a proposed solution. In addition, behavioural elements inside cases are accessible in this notation and so elements of a case can be used to construct a new solution.

The revision phase relies on the simulator and theorem prover (see section 7). By using a simulator, we automatically verify whether the proposed solution covers the behaviour exemplified in the input. If CABS discovers any discrepancies at this semantic level, it makes an attempt to adapt them, or points out where the differences lie, and requests further refinement of the input. The theorem prover may also be used to identify parts that need adaptation.

Finally the user can use the simulator and the theorem prover to explore whether the new formalised service meets his intention. If not, he provides more input examples, or, if his idea of the behaviour has changed, he modifies the previously given input examples. A confirmed solution is then stored in the case library (if it has been successfully designed and implemented), which bring us back up in the right upper corner of figure 2.

4. A Simple Case Description Logic

To represent cases, we have chosen a simple logic based on first-order predicate logic extended with a frame axiom [10,7,13]. Note that since we are using our coarse-grained specifications only as a means of identifying the appropriate designed and implemented services rather than modelling all the details of services, it is sufficient to use a comparatively simple logic. Simplification gives us further advantages by making the specification more accessible to users. For example, it is easier to state behavioural sequences because we ignore problems arising from asynchronous events. It is also possible to provide tractable methods for interfacing to the logic via natural language and/or graphical systems (see e.g. [8, 5, 6, 25, 9]). The behaviour sequences may also be used to test the final specification and for test generation for the final implementation, in addition to tests generated form the specifications [24].

The logic represents transition rules that handle changes, and intra-state rules that handle domain knowledge inside states, i.e. facts true at time t_i (see figure 3). A frame axiom moves all unchanged facts from the previous state T, to the next one, T+1. A transition rule is constructed using two types of term:

o(T, E) denotes that an event, E, occurred at time T.
p(T, P) denotes that the property, P, holds in the state of the system at time T.

Preconditions of transition rules must contain a single, triggering event and may also contain a conjunction of system properties (or their negation) which determines whether the transition rule can apply to the current system state. The conclusion of a transition rule contains a conjunction of properties (or their negation) which will hold in the succeeding state after the transition rule is applied. An example of a transition rule is given below, where offhook is the stimulus signalling that the user has lifted his/her receiver:

```
Transition Rule: normal_offhook
   ∀ SubscA
   o(T+1, offhook(SubscA)) &
   p(T, idle(SubscA)) &
   ¬p(T, ∃ SubscB calling(SubscB,SubscA))
         →    ¬p(T+1, idle(SubscA)) &
               p(T+1, dialtone(SubscA)).
```

Stimuli are sequenced in order to simplify the logic: we do not attempt in this high-level specification to specify what should happen when signals are competing (e.g. if two users call a third user at exactly the same time), and we suggest that the decision of how to resolve such situations is not necessarily a requirements choice, and can be dealt with in the design process. Figure 3 shows the model used in the formal requirements specifications of telecommunications services. Sequences of stimuli which are provided by users of telephones are used to activate appropriate

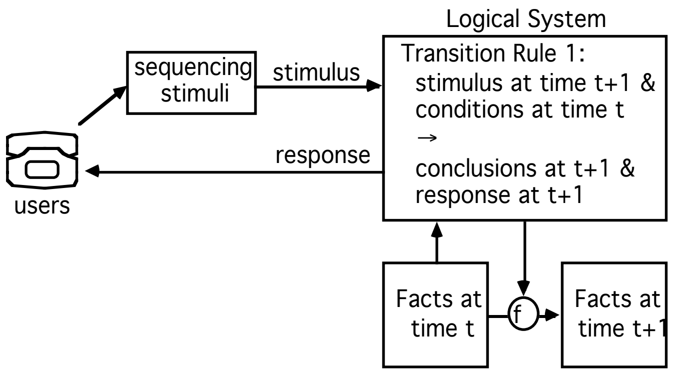

Fig. 3. Model of the dynamic behaviour of telecommunications network.

transition rules. As a consequence, a sequence of states is generated, containing sets of facts that describe the system after each event (*f* represents the frame axiom, which transfers unaltered facts from the previous time *t* to the current time *t+1*).

Because of the simplifications which we have made in our high-level specification language, we are able to simulate the behaviour obtained from these specifications by using a fairly simple theorem prover and simulator (see section 7). The theorem prover and simulator have been implemented in Prolog, together with a basic environment which allows the designer to test the specification and refine it in accordance with her ideas. It is necessary that this process be manual since we cannot know what the designer has in mind. We cannot require that she make a complete formal and correct description of her ideas in one step. Most likely, she will refine her ideas and give them a formal representation after she has simulated the formalised behaviour.

5. Assigning Behavioural Features to Cases

In the telecommunications domain, it is natural for users to describe new services by giving examples of the behavioural sequences that they should produce. The task of our case-based system is to locate existing services which most closely match these behavioural examples, based on their high-level specifications [17]. Since our case library consists of sets of transition rules, we must provide a means of matching these rules to behavioural examples. The behavioural examples may be given in a variety of notations, such as restricted natural language, graphical notations, scenarios etc., as long as they can be translated to a set of partial transition rules. We have chosen an intermediate formal representation, used as a starting point in producing a set of transition rules capturing the behaviour. The following is a behavioural example in its intermediate notation accompanied by a translation into English:

phoneNumber(A,111) &	A´s phone number is 111.
phoneNumber(B,222) &	B´s phone number is 222.
idle(A) &	Subscriber A is idle.
idle(B)	Subscriber B is idle.
–	then
offHook(A)	A lifts his receiver.
->	and as a consequence
dialTone(A)	A hears a dial tone.
–	then
dials(A,222)	A dials 222.
->	and as a consequence
ringTone(A)&	A hears a ringing tone.
ringSignal(B)&	B hears a ringing signal.
..	..

Table 1. Intermediate notation and translation to English.

A behavioural example starts with a conjunction of terms denoting the main features classifying exemplified states in which the following event (after the symbol "–") occurs. Thereafter a sketch of some of the terms outlining the main characteristics of the resulting state are given (after the symbol "->"). The last two steps may be repeated.

It is easy to generate a set of rules that precisely covers the behaviour given in a behavioural example. However, what we want is a set of rules that covers general behaviour, without excluding all other behaviour. Since humans often leave out obvious statements, we may wish to add some of the assumed domain knowledge. For this and for handling instances and variables we need some heuristics. This is acceptable, since the transition rules generated from the input are mainly used as indexing features in the matching process. The original input examples are also kept in their initial form, to be used later in validation and verification.

6. Re-Using and Finding the Best Matched Case

In section 5 we described how to put behavioural examples into rule form in preparation for the matching process. We shall now sketch the matching algorithm itself. Our aim is to reuse as much as possible of previously specified formal requirements, which is possible if the new demands on the system are semantically similar to previous demands and the previous demands are adaptable [26].

We treat the stimulus, condition and conclusion elements of the transition rules as sets of atomic terms. With this approach it is easy to identify matching rules. Figure 4 gives examples of different matches of elements from transition rules in the case library, and partial rules.

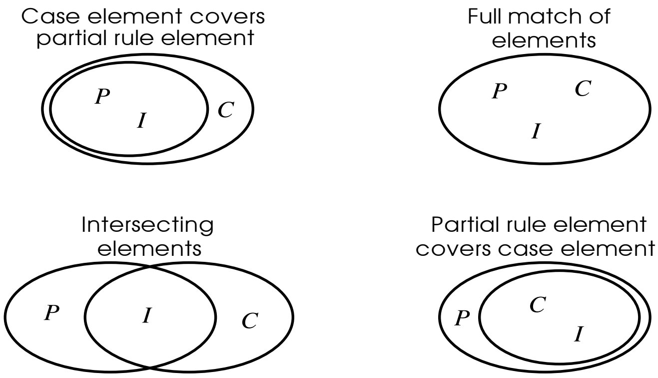

Fig. 4. Examples of different matches of transition rule elements.

An element in a rule is either a stimulus, condition or conclusion element. An element is a set of terms (a stimulus element is always a set with only one term). *P* denotes an element in a partial transition rule. *C* denotes the corresponding element of the case library rule. *I* stands for the intersecting terms in these two sets of terms. The relation between *P*, *C* and *I* is used as a basis for the final scoring. An example of two condition elements from two transition rules are:

Condition element in rule P_i:□p(T,idle(X)), p(T,last_diald(X,Nr))
Condition element in rule C_j:□p(T,redirect(X,Nr)), p(T,idle(X))

The intersection *I* is in this match a set containing one term, {p(T,idle(X)}. The following is an example of the process from behavioural example to a scored match:

Behavioural example (the notation for which bears similarities to signalling schemes for informal telephone service examples, see e.g. [18]):

idle(A) – offHook(A) -> dialTone(A) & idle(B) – dials(A,222) -> ringTone(A) & ringSignal(B).

Translating this into two partial rules gives:

```
Partial Transition Rule P1: ∀ A
                          o(T+1,offHook(A)) &
                          p(T,idle(A))
                             →  p(T+1,dialTone(A)).

Partial Transition Rule P2: ∀ A NR B
                          o(T+1,dials(A,NR)) &
                          p(T,idle(B))
                             →  p(T+1,ringTone(A)) &
                                p(T+1,ringSignal(B)).
```

In most cases this translation is a straightforward process, but in some situations heuristics are used to make an assumption about what the user means or to keep the partial rules within the restrictions opposed on transition rules. This can be accepted since we mainly use the partial rules to index the case library. One heuristic is used to avoid introducing unbound variables in a transition rules conclusion. In the above example this is noted by the fact that idle(B) in the behavioural example is only used as a precondition in P2. If it had also been used as a conclusion in rule P1, we would have introduced an unbound variable in the conclusions, which would conflict with our restrictions.

C1 and C2 are two transition rules (oversimplified to focus the attention on matching) belonging to the case “standard telephone call”:

```
Transition Rule C1: call busy
              ∀ SubscA NR SubscB
              o(T+1,dials(SubscA,NR)) &
              p(T,answers_on_number(SubscB,NR)) &
              ¬p(T,idle(SubscB))
                   →  p(T+1,busy_tone(SubscA)) &
                      p(T+1,call_busy(SubscA,NR)).

Transition Rule C2: normal call
              ∀ SubscA NR SubscB
              o(T+1,dials(SubscA,NR)) &
              p(T,answers_on_number(SubscB,NR)) &
              p(T,idle(SubscB))
                   →  p(T+1,ringTone(SubscA)) &
                      p(T+1,ringSignal(SubscB))&
                      p(T+1,last_called_nr(SubscA,NR)).
```

Only the matching of P2 is illustrated. We start by comparing P2 with C1. Some parts of the rules so standardised that they can be excluded in the matching process (e.g. time information and quantifiers). All three elements (stimulus, condition, conclusion) are to be matched, we start with the stimulus element:

Stimulus element in P2: dials(A,NR)
Stimulus element in C1: dials(SubscA,NR)

This gives a full match (identifying appropriate variables in the two terms).

Condition element in P2:idle(B)
Condition element in C1: answers_on_number(SubscB,NR),
¬idle(SubscB)

The condition element in the rule C1, from the case library, has a negated form of an expression in P2. At this stage we simply conclude that the two sets do not have any common terms and contain one negation (used later when elements are finally ranked). The intersection of the conclusion element in C1 and P2 does not contain any elements, hence P2 and C1 only have a full match in their stimuli.

We now continue by matching P2 with C2, where again their stimuli match fully. We then compare their conditions.

Condition element in P2:idle(B)
Condition element in C2: answers_on_number(SubscB,NR),
idle(SubscB)

The condition of rule C2, from the case library, covers the condition of P2, hence the condition of C2 is more restricted than P2. Similarly the conclusion element of C2 contains one additional conclusion term.

Conclusion element in P2:ringTone(A), ringSignal(B)
Conclusion element in C2:ringTone(SubscA),
ringSignal(SubscB),
last_called_nr(SubscA,NR)

Finally we employ a heuristic scoring algorithm to produce a numerical triple for each match and sort the matching rules in the case library, 'best first'. The approach taken is to give a percentage figure to each matching element in the rules (compare figure 4 with table 2).

P2 matching C1:

The intersection I is 100% of C1-stimulus and 100% of P2-stimulus.
The intersection I is 0% of C1-condition and 0% of P2-condition.
A negation of a term exists.
The intersection I is 0% of C1-conclusion and 0% of P2-conclusion.

P2 matching C2:

The intersection I is 100% of C2-stimulus and 100% of P2-stimulus.
The intersection I is 50% of C2-condition and 100% of P2-condition.
The intersection I is 67% of C2-conclusion and 100% of P2-conclusion.

Table 2. Coverage percentage of intersection for C1 and C2.

The fact that there exists a negation of a term in the match of the P2-condition and the C1-condition indicates that it is a mismatch, hence C1 may be excluded from further calculations. In CABS the user can decide whether or not to apply this filtering criterion to negation. Comparison of the scores for the three individual elements of the match provides us with a final ranking of each rule.

The pseudo-code for the matching algorithm appears below (all the domain-specific parameters have been omitted):

For all partial transition rules generalised from the input P_n:
    For all transition rules in the case library, C_m:
        For p in {stimulus element, condition element, conclusion element}:
            Calculate the intersection, I_p for P_{np} and C_{mp}
            Calculate the coverage percentage of I_p on P_{np},
            Calculate the coverage percentage of I_p on C_{mp}.
        Determine the final score for C_m by:
            Apply filtering criterion to negation (if a negated term exists, we may either choose to ignore it or to weight the result, depending on how the user has parameterized the system) in order to get a final score for C_m as a match for P_n.
For all cases (requirements specifications), S:
    For all partial rules P_n:
        Take the score from the rule in S which has the best score as a match for P_n and use it in order to score S in total.

7. Revising the Proposed Solution

CABS performs four steps of revision (see figure 2): test proposed solution against input (simulator) and against general domain knowledge (theorem prover); adapt any differences or ask user for clarification (by providing or refining input examples); finally the user explores the proposal with the simulator and theorem prover and confirms the behaviour or refines/modifies his input examples.

To perform a verification between the input examples and the proposed solution, we simulate the proposed solution and use the intermediate input as input to the simulation. If the proposed solution covers the behaviour of the input, the next step is to prove general domain properties about the solution. Examples of such properties in telephone services may be:

1. A subscriber cannot be in speech connection with himself.
2. In all situations a subscriber should be able to request to leave the current service (on_hook).

For the purpose of refining and testing the requirements specifications, a user interface is provided for the simulator and theorem prover. The simulator allows the user to give sequences of stimuli and evaluate whether the response exhibited by the formal specification corresponds to his intentions. This step is important in refining the designer's idea of how the service should behave in its final state. If the service does not correspond to his intention, the user has to provide more input examples, or refine previous given input examples. One other advantage of simulation, compared with theorem proving, is that it is more resistant to inconsistency in the formalised requirements, which is to be expected during the refinement process.

An example of a simulation is given in figure 5. We first display the initial facts in our simulation. If we want to simulate subscriber a1 going off-hook at time 1, we type O(1,offhook(a1)). The simulator triggers all the rules with offhook as their triggering condition and with all their conditions true, and thereafter shows the result (facts at time 1). To check if our set of transition rules behaves as expected if subscriber a1 is calling herself, we give the stimulus O(2,dialling(a1,111)). If a term is crossed out, it is not true at this time, but was true in the previous state. If a term is shown in bold face, it is a new term that has been added.

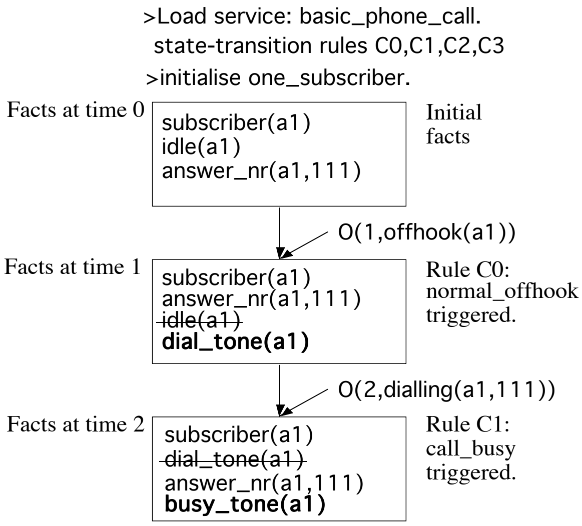

Fig. 5. Example of a simulation

The simulation system is highly interactive with the designer because full expansion of all possible states would require a huge amount of memory in any non-trivial specification. Fortunately, expanding the search space interactively by only a limited number of steps at a time is already of value in our domain since phone users are not usually expected to take part in any complex sequence of actions before returning to the initial state (hook on). Proving that a particular state cannot be reached in any sequence of, say, 8 steps will therefore be desirable for the user and will help him to validate his formal requirements specifications.

8. Related Work

Producing formal requirements from informal ones is an active research area. Much research in formal methods aims at producing detailed specifications of software, and the level of detail and difference in abstraction between the specification and the software to be produced is often small. We have adopted the approach of highly restricting the formalism and only outlining the main behaviour in our requirements specifications. With this approach we avoid some of the problems of using formal specification. There are similarities between the CABS approach and systems such as ARISE [6], AIR [20] and WATSON.

WATSON [17] is in essence based on the same ideas as CABS and has influenced our research. WATSON also starts with scenarios (similar to behavioural example sketches) which are used in a variety of different ways to produce formal requirements of telephone services. WATSON uses extensive domain knowledge (about telephone hardware, network protocols, preferred styles of control skeleton design, etc.) in the process. The user is consulted to resolve problems that the system cannot resolve itself. One of the key differences is that CABS takes the approach of using examples to generate features in order to identify and reuse previously specified services. WATSON has proved to be difficult to scale up for realistic use [24]. Our belief is that case-based reasoning and extensive reuse of previous requirements may help to bridge the gap between informal requirements and formal requirements and aid in the task of updating a previous implementation to capture the new requirements.

There has been much research in the area of applying case-based reasoning to the domain of design. Examples of systems using case-based reasoning to tackle such complex tasks are CADET [27], BOGART [21], DEJAVU [2], KRITIK [14] and SUPPORT [23]. A number of different approaches are used, such as multi-level representations, verifying results by qualitative simulations, and derivational analogy (i.e. the storing and reuse of design plans). In particular, case-based planners explore the use of formal logical representations. NETTRACK (Network Traffic Management Using Cases) [3] is a system which uses formal logic and which originally used a representation similar to situation calculus related to the representation used in CABS. CABS narrow focus on a particular class of specifications allows us to automate the case-based reasoning process considerably, compared with more general systems.

9. Conclusions

We have presented a system that produces formalised requirements, capturing the dynamic behaviour of a particular class of requirements (sequential, non distributed, deterministic). Given an outline of a required behaviour, the system produces a formal requirements specification capturing certain dynamic aspects of the requirements, constructed from previous cases and parts of cases. Selecting the way in which cases should be represented is an essential aspect of providing case-based support for specification of system requirements. A case should be able to represent the dynamic behaviour that the specification calls for. In our approach, the requirements designer has only to give examples of a new service's behaviour. The input is translated to a representation, more suitable for matching, and a set of generalised partial transition rules is produced. These rules are then used in the matching process, and modules with similar behaviour are identified by means of a simple and sufficient matching algorithm based on set theory.

The logic used for the representation of cases contains transition rules, terms (stimuli, responses, facts) and a frame axiom handling change of time (discrete time steps). This logic has proved to be sufficient for outlining and testing (by simulation and theorem proving) the behaviour of some telecommunications services' coarse-grained behaviour [12]. The combination of simple representation and a case-based approach that we advocate can be successfully applied in order to reuse elements of earlier requirements. Because the older cases describe the behaviour of existing services that have been fully tested, integrated and implemented, the effort required to integrate a new service with these other services - or to test it - is considerably reduced.

If the behaviour of a case does not fully conform to the behavioural examples, the missing elements of behaviour can be filled in by using the rules, generalised from the examples. This will produce a naive solution - which the user may subsequently refine and test - that conforms to the behavioural examples. Since the logic is comparably simple, it is easy to verify by simulation that the identified service and its transition rules correspond to the behaviour in the input examples. The user can also simulate the system's behaviour to examine other behaviour which he may not have stipulated in the original examples, but which may have arisen as a result of reuse (such as interaction with other services).

References

☐ 1. A. Aamodt, E. Plaza, Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches. *AI Communications*, vol. 7 no. 1, 39-59, 1994.

☐ 2. T. Bardasz, I. Zeid, Dejavu: A Case-Based Reasoning Designer's Assistant Shell. *Artificial Intelligence in Design '92*, J.S. Gero (ed.), Kluwer Academic Publishers 477-496, 1992.

☐ 3. R. Brandau, A. Lemmon, C. Lafond, Experience with Extended Episodes: Cases with Complex Temporal Structure. *Workshop on case-based reasoning*, Morgan Kaufmann, 1-12, 1991.

☐ 4. A. Bundy: Tutorial notes: reasoning about logic programs. *Second International Logic Programming Summer School, LPSS '92. Proceedings,* G. Comyn, N.E. Fuchs, & M.J. Ratcliffe (eds.), Springer-Verlag, 232-277, 1992.

☐ 5. H. Dalianis: Aggregation in the NL-generator of the VIsual and Natural Language Specification Tool. *The Sixth International Conference of the European Chapter of the Association for Computational Linguistics, EACL-95*, Dublin, Ireland, 1995.

☐ 6. E. Davis: *Representations of Commonsense Knowledge*, chapters 2 and 3. Morgan Kaufmann, 1990.

☐ 7. J.-P. Echarti, G. Stålmarck: A logical framework for specifying discrete dynamic systems. *Technical Report*, Ellemtel Telecommunication Systems Laboratories, 1988.

☐ 8. M. Engstedt: A Flexible Specification Language using Natural Language and Graphics. *MSc thesis*, University of Edinburgh, 1991.

☐ 9. N. Fuchs, R. Schwitter, Specifying Logic Programs in Controlled Natural Language. *Workshop on Computational Logic for Natural Language Processing,* Edinburgh, 1995.

10. P.J. Funk: Development and Maintenance of Large Formal Specifications Supported by Case-Based Reasoning. *Technical Report TP026*. University of Edinburgh, 1993.

11. P.J. Funk, D. Robertson: Requirements Specification of Telecommunication Services Assisted by Case-Based Reasoning. *The 2nd International Conference on Telecommunication Systems, Modelling and Analysis*, Nashville, 160-169, 1994.

12. P.J. Funk, S. Raichman, ROS, an Implementation Independent Specification for ISDN. *Technical Report*, Ellemtel Telecommunication Systems Laboratories, 1990.

13. M. Gelfond, V. Lifschitz: Representing action and change by logic programs. *Logic Programming*, 301-321, 1993.

14. A.K. Goel, Representation of Design Functions in Experience-Based Design. *Intelligent Computer Aided Design*, Elsevier Science Publishers, 283-303, 1992.
15. A. Hall: Seven Myths of Formal Methods. *IEEE Software,* September, 11-18, 1990.
16. W.L. Johnson, K.M Brenner, Developing Formal Specifications from Informal Requirements. *IEEE Expert*, vol. 8, no. 4, 1993.
17. V.E. Kelly, U. Nonnenmann: Reducing the Complexity of Formal Specification Acquisition. *Automating Software Design*, M. Lowry, & R. McCartney (eds.), 41-64, 1991.
18. S. Klusener, B. Vlijmen, A. Waveren: Service Independent Building Blocks-I; Concepts, Examples and Formal Specifications. *Technical Report P9310*, University of Amsterdam, 1993.
19. J.L. Kolodner: *Case-Based Reasoning*. Morgan Kaufmann (1993).
20. N.A.M. Maiden, A.G. Sutcliffe, Requirements Engineering by Example: an Empirical Study. *Proceedings of IEEE International Symposium on Requirements Engineering*, 104-111, 1995.
21. J. Mostow, M. Barley, T. Weinrich, Automated reuse of design plans. *Artificial Intelligence in Engineering*, vol. 4, no. 4, 181-196, 1989.
22. K. Nakata: Behavioural Specification with Nonmonotonic Temporal Logic. D. Finn (ed.), *Preliminary Stages of Engineering Analysis and Modelling Workshop*, AID '92, 41-45, 1992.
23. Y. Nakatani, M. Tsukiyama, T. Fukuda, Engineering Design Support Framework by Case-Based Reasoning. *ISA Transaction*, vol. 31, no. 2, 235-180, 1992.
24. U. Nonnenmann, J.K. Eddy, KITSS - A functional Software Testing System Using a Hybrid Domain Model. *IEEE*, 136-142, 1992.
25. S. Preifelt, M. Engstedt, Results from the VINST Project (In Swedish). *Technical Report*, Ellemtel Telecommunication Systems Laboratories, 1992.
26. B. Smyth, M.T. Keane: Retrieving Adaptable Cases. In: S. Wess, K.-D. Althoff, & M.M. Richter (eds.), *Topics in Case-Based Reasoning*, Springer-Verlag, 1994.
27. K. Sycara, D. Navin chandra, R. Guttal, J. Koning, S. Narasimhan, CADET: A Case-Based Synthesis Tool for Engineering Design. *International Journal of Expert Systems*, vol. 4, no. 2, 167-188, 1992.
28. P. Zave, Feature Interactions and Formal Specifications in Telecommunications. *Computer*, vol. 26, no. 8, 1993.