Automated Test Case Generation from Use Case Specifications

Markus Lepper

Technische Universität Berlin/ISTI/ÜBB
lepper@cs.tu-berlin.de

Abstract Automated test performance and automated test case generation have been topics of increasing interest in academic research and are of high relevance to industrial practice.
Based on the analyzation of existing industrial documents and habits, the paper presents (1) a modeling language for requirement engineering, giving exact semantics to the well-known, bur usually only imprecisely specified „Use Case“ approach, (2) a practical-oriented architecture for integrating well-founded formalisms into existing semi-formal documents, and (3) an algorithm for deriving test cases with full coverage properties by means of constraint resolution.

1 Introduction

1.1 Use Case Specifications: Industrial Practice and Academic Research

Describing a system's behavior by filling in *Use Case Templates* is a widespread means in industrial practice, — during the requirement specification phase as well as in early system design steps.

This specification technique can be called *semi-formal*: The overall system specification (or description) is given by a set of files or text documents, each of them describing one single Use Case, i.e. one *cycle of interactions* between two or more subsystems.

Below this level, the text documents are structured according to some (in-house) Use Case Template, which defines the fields to fill in, — firstly for meta-information like a key name, list of the authors, editing history, short description, comments etc.

Central substance of such a use case description is a (more or less formally encoded) specification of a family of sequences of interactions between the involved subsystems. In practice this is mostly notated as a simple ms-word (et.al.) listing format containing natural-language descriptions of the interactions. This specified set of sequences is called the „good path“, because it describes the required system's behavior in the absence of errors or interruptions.

Normally a *separate* section is foreseen for „ugly paths“, i.e. the interaction sequences happening in response to some error, malfunction or user interruption [1].

[1] We found even templates where all *timing requirements* formed a separate section. For each timing requirement the event of the „good path“ it was meant to be related to, was only indicated informally, by natural language. But, on the other hand, all those templates which relate ugly paths and timing constraints to the steps of the good path somewhat more exactly by referring to the automatically generated item numbering in the „list format“ of the „good path“ are not robust against editing the latter!

There are ongoing research activities on the industrially oriented level of software-engineering, discussing template design, standardization etc. [Col98], [Ber98].

This level of structure will be treated in this paper, and must not be mixed up with the next layer above, represented e.g. by the UML Use Case *Diagrams*. These represent an algebra „on top“ of the single Use Case definitions, and have been explored mathematically in [BGK97] and [Ste01].

1.2 Benefits and Problems when Introducing “Formal Methods”

The theoretic benefits for using „Formal Methods“ are widely accepted in academia as well as in industry. Let us define „using formal methods“ as simply using (for specification and modeling purposes) only languages with a mathematically precisely defined semantics[2]. Doing so, (suddenly and magically) it *may* become possible to perform automated coverage analysis, automated transformation, simplification and visualization, symbolic computation, reasoning and proving etc. on the specifications and models.

In practice there are severe obstacles against an effective introduction of „better tools“:

- Disturbance of ongoing production processes must be minimalized.
- There must be acceptance and easy learning for new tools.

The central requirements for success in introducing formal methods into industrial processes seem to be (1) maximal adaptability to existing formalisms, methods, tools and *habits*, — and (2) personal acceptance of tools and languages by the engineers, — again requiring adaptability. So the idea rose quite naturally to establish a *gradually refinement* from existing semi-formal industrial use-case description documents to well-founded formal specifications.

Since Use Case documents are mainly text-based, this met luckily with the politics of the author and his colleagues, to support the application of „formal methods“ by supplying tailored, domain-specific languages to the practitioners. and fits well to the concept of *multiparadigmatic design*, as presented in [PCDG00].

2 The useCASE Approach

In the following we present an abstract and generic architecture called useCASE, designed to support such a gradual refinement. It is the result of a small project in meta-specification, carried out with partners and customers from industry, to capture the experience and results of analysis into an abstract system architecture[3].

[2] Like DAVID L. PARNAS said, „Why call it ’formal methods’? It’s just applying mathematics!“, —panel discussion at IFCEM’00, York.

[3] Since the level of description is highly abstract in the following, and the results may seem to be rather „theoretical“, we want to mention explicitely that indeed all decisions on the *detail* level try to be a „condensed concentrate“ from the analyzation of existing, practically used industrial specification documents, and intensive discussions with engineers, developers as well as specifiers.

Due to limited resources of this design project only small parts could be implemented, but large parts, including the central functionality, had already been implemented in pre-going projects, by others and the author.

The paper centers of test case derivation by constraint resolution and the presentation of an adequate specification language $\mathcal{L}_S^2$, so further results of the useCASE project are shortly described in an appendix.

Please note that the following is written in firstly from the viewpoints of System Architecture and Language Design. In the field of Constraint Resolution the author is just a user, though qualified enough to judge that the constraint solvers required by the useCASE architecture must be of „cooperating“ style, as in the integrating approach of [Hof01].

2.1 Foregoing projects

The useCASE concept is based on the experiences and results of foregoing common project of TU-Berlin/ÜBB and daimlerChrysler FT3/SM. In this context the basic problems of semantics, of translating domain languages into an executable model, and executing test protocol evaluation, have been solved [GL00]. Two case studies had been modeled, an Automated Teller Machine (just two agents, a human and the ATM) and an elevator (a multi-agent system, since more than one user are supported).

This had been done using the ZETA system, which is an outcome of the ESPRESS project ([BG99],[BG98]). The use case domain languages were translated to terms in *Executable Z*, which were evaluated by the ZAM, a machine designed and realized by WOLFGANG GRIESKAMP, which can evaluate a large subset of Z expressions [Gri99,Gri00]. Recently this approach has been carried over to the semantics of AsmL [GLST01].

2.2 Applicability of our Approach

The useCASE approach and architecture is applicable to all behavior specifications, which come in form of specifications of finite sequential operations, and fulfill the following structural properties:

1. All operations will (at last) be represented by some „physical“ interaction, i.e. some method call, PDU exchange or electrical action which is detectable by an automatic system.
2. The structure of the system’s behavior must be cyclic and contain *one single „Global Idle“ state*. It is a widespread opinion on the semantics of Use Cases, that such a „single global idle“ is the state taken by the system in the absence of any external stimulus, and that after any stimulus the system will return to „global idle“ in finite time.
3. Specification and Test are both pure *Black Box* style. There is no knowledge on any „inner state“ of any subsystem. The „state“ managed by the formalism $\mathcal{L}_F^2$ presented below is just an *observation state*, i.e. an auxiliary memory into which the *observing* process can store some historic data on its passed observations.

2.3 Auxiliary Language Modules

The requirement for adaptability, as mentioned above, led to the basic design of a collection of „layered languages“, i.e. different „modules“ or „aspects“, which can be plugged together to form a tailored specification language to fit into an established industrial context.

One possible partitioning is (cf. figure 1) :

- $\mathcal{L}_F$, a layer which provides the notions of basic and universal mathematics, called „foundation language“.
- $\mathcal{L}_S$, the central specification calculus.
- $\mathcal{L}_D$, the overall document structure seen as a „language“.
- $\mathcal{L}_P$, the definition of the „physical“ information which is to be monitored in an actual test environment.

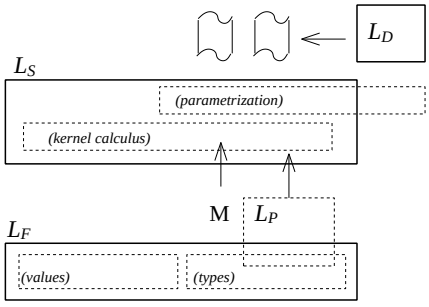

Figure1. Layered Languages

Each instance of a „foundation language“ $\mathcal{L}_F$ represents a basic mathematical toolkit, serving as a basis to give semantics to varying, most different upper layer instances. Each $\mathcal{L}_S$ represents a small „specification grammar“, which may vary with projects and applications. With $\mathcal{L}_D$ the „legacy habits“ of representing specifications (and reports etc.) as disk files, data base entries, printed copies etc. is modeled as a „language“. This is necessary for integrating the semi-formal documents into an executable program. Finally $\mathcal{L}_P$ defines the mapping of the logical world of test data messages to the „physical“ bus messages, hardware signals etc., which are produced or received by the involved subsystems.

Foundation Language $\mathcal{L}_F$ Each instance of a „foundation language“ has to provide

- a **type language**,
- a corresponding **value language**,
- a set of **predefined** types and values,
- together with corresponding predefined **functions**, partly written as **operators** in the value language.

The most important demand to this layer is *political* : The mathematical language foundation language $\mathcal{L}_F$ must be accepted by all participants of a project. So „seen from **outside**“ it should resemble a kind of „common high school mathematics“, the notations should be somehow known, the semantics should be explainable in a few words etc.

Seen from the **inside** all this is merely impossible: The task of giving well-defined semantics (in the strict mathematical sense) to any language is a wide field of research on its own, hosting hundreds of publications, the results of which can hardly be explained in a few words.

$[V_0]\ [T_0]$

$$
\begin{array}{lcl}
i & : & \mathit{Ident} \\
\\
T ::= \mathcal{T} & = & \mathtt{SET}\ T \mid \mathtt{SEQ}\ T \mid \mathtt{MAP}\ T_1\ T_2 \\
& \mid & \mathtt{DATA}\ (\ i_n\ T_n\ )^+ \\
& \mid & \mathtt{integer} \mid \mathtt{float} \mid \mathtt{text} \mid \ldots \\
& \mid & T_0 \\
e \in E & ::= & \mathit{built\text{-}in\ denotation} \\
& \mid & V_0 \\
& \mid & \mathtt{if}\ p\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \\
& \mid & e_1\ \ (\mathtt{+} \mid \mathtt{-} \mid \mathtt{*} \mid \ldots \mid \cup \mid \cap \mid \ldots)\ \ e_2 \\
& \mid & \{\ e_1\ , \ldots ,\ e_n\ \bullet\ p\ \} \\
& \mid & \langle\ e_1\ , \ldots ,\ e_n\ \bullet\ p\ \rangle \\
& \mid & \{\ e_1 \mapsto e_1', \ldots ,\ \ e_n \mapsto e_n'\ \bullet\ p\ \} \\
& \mid & [\ i_1 \mapsto e_1, \ldots ,\ \ i_n \mapsto e_n\ \bullet\ p\ ] \\
p \in P & ::= & \mathtt{false} \mid \mathtt{true} \\
& \mid & e_1\ \ (< \mid \leq \mid \ldots \mid \subset \mid \in \mid \ldots)\ \ e_2 \\
& \mid & p_1\ \ (\wedge \mid \vee \mid \ldots)\ \ p_2 \\
\\
\{\mathtt{false}, \mathtt{true}\} & \subset & E_{VALUE} \subset E \cup P
\end{array}
$$

Figure2. Example Syntax for a Basic Foundation Language $\mathcal{L}_F^1\,[V_0, T_0]$

In an application-oriented project like USECASE this task can only be accomplished by *mapping* the semantics of an $\mathcal{L}_F$ onto some well explored formalism like Z or AsmL. But further understanding of these formalisms also requires deep considerations w.r.t. axiomatization, logic, definedness etc.

Luckily in **practice** this dichotomy hardly matters: Consider e.g. the little foundation language $\mathcal{L}_F^1$ as given in figure 2, which has been designed for the special needs of a certain industrial project, but still is of sufficient generality.

The pure syntax, together with some „rough“ approximation to the semantics, can be explained to any developer in a few minutes. *We* as implementors, and not the applicants, have the problem that the pure syntactic definitions are much too rich, — that the language has to be restricted to an executable subset, which has to be identified and recognizable[4].

The pure syntax alone would of course allow the applying engineer to write down e.g. „self-applications“ and „Russel-sets“, but he or she will mostly not intend to do so! Only indirectly, in the transitive closure of some (probably erroneous) circular definition, these anomalies can appear in daily practice.

The type language of $\mathcal{L}_F^1$ contains

- The type constructors SET_, SEQ_ and MAP_,_ .
- A means for defining *Free Types*.

The given types should include (at least) one Integer and one Floating Point type, and some kind of „Text“ or „String“ type.

At this point one sees that $\mathcal{L}_F^1$ is not a really „abstract mathematical“ language, but some decisions were taken due to *pragmatical* reasons :

- It would have been much „cleaner“ to treat „Free Types“ as syntactic sugar, realizing them by maps or by sequences or by introducing a plain product type constructor.

[4] The field of typing could not be worked upon in the scope of this project. Any *type system*, which should be implemented for such a front-end, is therefore primarily needed for *restriction* of the pure syntactical language, e.g. forbidding free types as leaf value types, thereby eliminating the problem of referential un-computability.

But the Free Type construct (1) can rather easily be explained to any domain expert who is acquainted to things like „`struct`“ in C or C++ or „`object`“ in Java, and (2) it can be used to describe both user defined „data structures“ as well as user defined „enumeration types“ rather naturally.
So it has been included as first class resident.

– Finite Mappings became first class resident due to the practical needs and habits of the engineers and due to the structure of the SUT (user lists had to be maintained, etc.).
– Allowing function-valued fields allows to describe behaviors in an object-oriented style. So second-order functions are allowed, higher order functions are not supported.

So also this foundation layer can *vary* with the habits and methods of the users, — but generally the number of the instance languages of this foundation layer should of course be kept as small as possible

Description of „Physical“ Encodings by $\mathcal{L}_P$ The separate position of the „Physical Language“ $\mathcal{L}_P$ is also due to pragmatic reasons: The data structures of the „physical entities“ which have to be specified and later observed (e.g. the datagram packages of a certain bus protocol, — sequences of abstract symbols representing man/machine interaction, — I/O activities on TTL pins, etc.) may vary much more frequently than the specification formalism. There may even coexist different of these „message formats“ in the same project.

In practice we want these definitions to be extracted automatically from the domain specific description languages, e.g. from ASN.1 text files, or from IDL data base entries, etc. At last $\mathcal{L}_P$ delivers nothing more than a Free Type definition for the atomic interactions. This type definition is used as a parameter when instantiating the specification languages.

Capturing the relation between documents by $\mathcal{L}_D$ If collections of separate documents containing semi-formal specifications shall be comprehended to a consistent specification, the relation between the different files (or other „physical“ entities) has to be lifted to the formal level.

For this purpose we propose the existence of a language layer $\mathcal{L}_D^{\square}$, which may and probably will vary with the projects. This language can be rather primitive and just has to establish the relations between the different, separated documents and map their contents to a formal notion of „scope“. E.g. there can be a set of „global“ documents defining overall needed functions and common sub-use cases, and one document for each single use case.

In the current design we simply foresee one single distinguished plain text file as „catalog document“, listing all specification documents and the globally visible „global definitions“ documents. All of the formers will simply be textually included in all of the latters. In general one would prefer some automated adaption layer for using some repository system.

Not before some semantics of „parameterizable modules“ is defined, (see below 2.4), implementing $\mathcal{L}_D$ becomes a real task, comparable to a „`make`“ system.

2.4 The Sequence Specification Languages $\mathcal{L}_S^n$

Opposed to the „minimalized“ layer $\mathcal{L}_F$, there has to be a multitude of instances of $\mathcal{L}_S$. They vary in different dimensions, e.g. the kind of domain to be specified, — the kind of tool into which to embed, — the data structures and communication topology of the given project, — existing notation conventions, e.g. the format rules of the existing semi-formal legacy data, — social habits, e.g. the „proprietary working slang“ established in the developing team, etc.

$[\mathcal{L}_F]\ [\underline{\alpha} :: \mathcal{FREETYPE}(\mathcal{L}_F)]$

$$
\begin{array}{lll}
M & \in & \mathbb{P}\ \underline{\alpha} \\
L & \in \mathit{Locations} & = \mathbb{F}\ \mathit{Ident} \\
K & \in \mathit{ObsState} & = \mathit{Locations} \nrightarrow E_{VALUE} \\
R & \in \mathit{Transition} & = \mathit{ObsState} \rightarrow \underline{\alpha} \nrightarrow \mathit{ObsState} \\
e & & : \ E_{VALUE}(\mathcal{L}_F)
\end{array}
$$

$$
\begin{array}{lll}
\sigma \in \Sigma & ::= & M \diamond R \\
 & | & \sigma\ \texttt{->}\ e\ \texttt{..}e\ \texttt{:}\ \mathit{interleave} \\
 & & \quad (\ \texttt{|}\ e\ \texttt{..}e\ \texttt{:}\ \mathit{interleave}\ )^* \\
\mathit{interleave} & ::= & \sigma\ (\ \texttt{!!}\ \sigma)^*\ (\uparrow\ M \diamond R)^*
\end{array}
$$

Figure3. Syntax of a Sequence Specification Language $\mathcal{L}_S^2$.

Since now the „use case language“ $\mathcal{L}_S$ is „pluggable“, we can install a syntax specially tailored for the needs of one distinct project:

$\mathcal{L}_S^n$: Basic Calculus Layer The example version $\mathcal{L}_S^2$ (see figure 3) has indeed been designed for a concrete industrial project, by abstracting from the real-world examples of semi-formal legacy documents.

The constructs of $\mathcal{L}_S^2$ are:

- $M \diamond R$
  Constructor for a basic event, combining a message pattern with an update on the observation state.
- σ_1 `->` e_1 `..` e_2 `:` i_2
  Sequentialization within given time window, — the first event of the following sequence (or interleaving of sequences) has to happen in the given interval after the foregoing event, i.e. the last event of the former sequence.
- σ_1 `->` e_1 `..` e_2 `:` i_2 `|` e_3 `..` e_4 `:` i_4 `|` ...
  Alternative paths.
- $(\sigma_{1,1}$ `->` ...$)$ `!!` $(\sigma_{2,1}$ `->` ...$)$ `!!` ...
  Interleaving of paths: The operational semantics can most easily be explained by a set of parallel processes (one for each sub-expression combined by the `!!`-operator) which merge their outputs into one single trace of events[5].
- $(i_1$ `!!` i_2 `!!` ...$)\ \uparrow\ M_1 \diamond K_1\ \uparrow\ M_2 \diamond K_2\ \uparrow \ldots$
  Exception mechanism: All traces which are prefixes of valid traces w.r.t. $(i_1$ `!!` i_2 `!!` ...$)$ can be ended (out of any depth of nesting) with an exception-indicating event from $\{M_1 \diamond K_1,\ M_2 \diamond K_2,\ \ldots\}$.

[5] Please note that this merging must include a transformation to achieve strict serialization of events, as required by the definition of *Trace* being a *function* from time into (always one single) event.

The definition of these constructors is derived from the analyzing of legacy use case templates: The notation supports timing windows as first class resident, — the „exception“ mechanism reflects the „ugly paths“ found in use case templates and allows to give more precise semantics by giving to them an exact „scope“.

The interleave operator ! ! has been successfully implemented in the ZetA-version [GL00]. There the semantics of parallelism had been chosen to be „CSP-like“, i.e. common message patterns (= „events“) were consumed by all parallel processes simultaneously.

It seems that in practice the interleave operator will mostly used on *top-level*, for combining independent use cases to one single system behavior[6].

Figure 4 gives the semantics of this language by defining a semantic function $[\![\_]\!]^T$ which assigns sets of traces and state transitions to syntactic constructs, and the fulfill-

$$
\begin{array}{lcl}
PDUs & = & \underline{\alpha}\,[E_{VALUE}(\mathcal{L}_F)] \\
T \in Time & :: & \mathcal{LINEAR} \cap \mathcal{TOTALORDER} \\
 & & \cap\, \mathcal{DENSE}
\end{array}
$$

$$
\begin{array}{l}
Trace == Time \nrightarrow PDUs \\
\forall\, T : Trace \bullet\ t_0 \in \mathrm{dom}\, T \\
\qquad \Rightarrow\ \exists \varepsilon > 0.0 \bullet \forall\, t_x |\ t_0 - \varepsilon \le t_x \le t_0 + \varepsilon \\
\qquad\qquad \bullet\ t_x \in \mathrm{dom}\, T \Rightarrow t_x = t_0
\end{array}
$$

$$
\begin{array}{l}
Fragments == Time \nrightarrow (PDUs \times Transition) \\
\forall\, F : Fragments \bullet\ (\lambda(a,b) \bullet a \mapsto b.1)(\!|F|\!) \in Trace
\end{array}
$$

$$
\begin{array}{lcl}
[\![\_]\!]^T & : & \Sigma \to Fragments \\
\in_F & : & Trace \times ObsState \leftrightarrow Fragments
\end{array}
$$

$$
\begin{array}{l}
[\![M \diamond R]\!]^T \ =\ \{\ t : Time, M' : PDUs \ \mid\ M \underline{\sim} M' \\
\qquad\qquad \bullet\ \langle\ (t, M, R)\ \rangle\ \}
\end{array}
$$

$$
\begin{array}{l}
[\![\sigma_1\ \texttt{->}\ i_2\ (\texttt{|}\ \pi)^?]\!]^T \\
\ =\ [\![\sigma_1\ \texttt{->}\ 0.00001\ \texttt{..}\ \infty\ \texttt{:}\ i_2\ (\texttt{|}\ \pi)^?]\!]^T \\
[\![\sigma_1\ \texttt{->}\ e_1\ \texttt{..}\ e_2\ \texttt{:}\ i_2\ \texttt{|}\ \pi\ ]\!]^T \\
\ =\ [\![\sigma_1\ \texttt{->}\ e_1\ \texttt{..}\ e_2\ \texttt{:}\ i_2]\!]^T\ \cup\ [\![\sigma_1\ \texttt{->}\ \pi\ ]\!]^T \\
[\![\sigma_1\ \texttt{->}\ e_1\ \texttt{..}\ e_2\ \texttt{:}\ i_2]\!]^T \\
\ =\ (\_ \cup \_)\ (\!|\ \{(a,b) \mid t_l - e_1 \le \mathsf{first}(b).1 \le t_l + e_2 \\
\qquad\qquad \wedge\ t_l = \mathsf{last}(a).1 \\
\qquad\qquad \wedge\ a \in [\![\sigma_1]\!]^T \\
\qquad\qquad \wedge\ b \in [\![i_2]\!]^T \\
\qquad\qquad \}
\end{array}
$$

$$
\frac{[\![\ i_1\ !\ !\ \ldots\ !\ !\ i_n\ ]\!]^T \ =\ I}{[\![\ i_1\ !\ !\ \ldots\ !\ !\ i_n\ \uparrow x_0 \uparrow \ldots \uparrow x_m\ ]\!]^T}
$$

$$
\begin{array}{l}
\ =\ \{t : Trace \mid\ \exists \hat{t} : Trace \bullet\ (\{\mathsf{last}(t)\} \lhd t) \frown \hat{t} \in I \\
\qquad\qquad \wedge\ \exists k \mid 0 \le k \le m \bullet \mathsf{last}(t).2 \underline{\sim} x_m\ \}
\end{array}
$$

$$
\begin{array}{l}
[\![\ i_0\ \texttt{!!}\ 1_1\ !\ !\ \ldots\ !\ !\ i_n\ ]\!]^T \\
\ =\ (\_ \cup \_)\ (\!|\ [\![i_0]\!]^T \times [\![1_1\ !\ !\ \ldots\ !\ !\ i_n\ ]\!]^T\ |\!)
\end{array}
$$

$$
\begin{array}{l}
\in_F : Trace \times ObsState \leftrightarrow Fragments \\
\hline
(t, o) \in_F F \\
\quad \Longleftrightarrow\ \exists f : F \bullet\ t.\mathsf{first}.1 = f.\mathsf{first}.1 \\
\qquad\qquad \wedge\ t.\mathsf{first}.2 \in f.\mathsf{first}.2 \\
\qquad\qquad \wedge\ o \in \mathrm{dom}\, f.\mathsf{first}.3 \\
\qquad\qquad \wedge\ o' == (f.\mathsf{first}.3)(o, t.\mathsf{first}.1) \\
\qquad\qquad \wedge\ (t.\mathsf{rest}, o') \in_F f.\mathsf{rest} \\
\quad \vee\ (t = \{\} \wedge \exists f : F \bullet f = \{\}\ )
\end{array}
$$

Figure4. Semantics of a $\mathcal{L}_S^2$.

[6] Please note that there is no general concept yet to give *timing constraints* on events of different interleaving paths.

ment relation $\in_F$, indicating if a given trace is part of the semantics of a given fragment (together with an initial observation state).

$\mathcal{L}_S^n$: Further Layers $\mathcal{L}_S^2$, as presented so far, is completed by further language modules, only the first of which is necessarily required. Not all of these modules have been subject of deeper research in the past useCASE project.

- Context Semantics, Bindings and Expression Evaluation.
  In $\mathcal{L}_S^2$, as presented so far, the sets of accepted messages „*M*“ and the transitions of the observation state „*R*“ are just given as abstract semantics, i.e. as pure mathematical objects. Still missing is a means for really *constructing* these objects.
  This is achieved by adding *locations*, and a certain notion of *context*. In such a context all expressions which contain identifiers (= names of locations) are evaluated by mere substitution as usual.
  Data construction could of course also be achieved by more abstract (therefore more versatile) means, e.g. a general fix-point semantics, but the cultural background of the personnel makes an „imperative“ approach more likely to be accepted.
  A compact encoding of this language layer will be discussed in 3.1.
- Abstraction.
  Of course the specifying engineer will need some means for abstraction. Our analysis showed that constructs like a parameterized „`GOTO`“ and „`GOSUB`“, i.e. continuation and non-recursive function calls, are sufficient and well-enough known to the engineers.
  Seen from the semantic aspect, this layer indeed introduces *functions*, which evaluate to specification formulae of our basic calculus.
  Implicit abstraction is introduced by implicit sharing, as e.g. in embedded or constructions, where different alternatives with identical continuation shall be describable by some notation like a `->(` b_1 | b_2 | ... `) ->` `c`
- Recursion / Inner Loops.
  There should be some combinators like those known from „Regular Expressions“ for convenient denotation of „inner loops“, as used in the example of figure 9.
  The same effect could be reached by the more general means of allowing recursive functions, which in most cases will be too strong a means: Important transformation and analyzing techniques become infeasible due to the power of the specification language.
- Modularization.
  This area partly overlaps with the $\mathcal{L}_D$ the language for document organization, see above 2.3.
  A consistent concept of modules and parameterization for use cases has still to be developed, strongly considering the results from research on the use case *diagram* level, like[BGK97], [Ste01].
  As soon as such a module concept is installed on the level of semantics, the document organization language $\mathcal{L}_D^($and any technical adapter to some repository) has to be implemented accordingly.

3 Test Data Generation by Constraint Resolution

For subset of $\mathcal{L}_S^2$ presented above, and of similar specification languages, it may be possible to generate test data automatically and to perform coverage analysis.

Several methods are known to gain information etc from behavior specification, cf. the survey given in [Sei01].

The one presented here supposes that all possibles paths through the specification (from „Global Idle“ back to „Global Idle“) can be unfolded into one single finite *tree* of possible system behaviors. This implies that all „inner loops“ are finite and limited at compile time, and that the cardinality increases linear with every „inner alternative“.

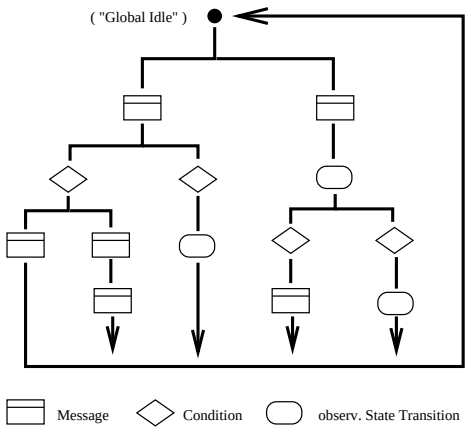

Figure5. A Behavior Specification seen in the linearized model

The principle is that for all „leaves“ of the tree of possible paths through the specification a collection of *constraints* is collected, which must be fulfilled for this path to be taken. Then for each possible path this constraint system must be resolved, i.e., one representing solution must be found, which will force the system to take this special path.

The main classification of paths under this concern is, if the decisions are solely controlled by the test environment, or are in-deterministically decided by the SUT. This issue will be presented in more detail at the end of this section.

3.1 Constraint Semantics

As mentioned above, the specification calculus $\mathcal{L}_S^2$ is enhanced by means for making assignments to locations of the observation space and for filtering message sets.

This can be depicted as applying to the grammar of $\mathcal{L}_S^2$ from figure 2 the substitution

$$M \diamond R \;\Rightarrow\; \mathit{EventDescription}$$

... in combination with the grammar ...

$$\begin{aligned}\mathit{EventDescription} == \underline{\alpha}\ &[\,!\,(\mathit{Ident}) \mid \mathit{Expr} \mid \_\,]\ \texttt{;}\\ &(\mathit{Ident}\ \texttt{::=}\mathit{Expr}\texttt{;} \mid \mathit{Pred}\texttt{;}\ )^*\end{aligned}$$

For ease of discussion and brevity of the algorithms we represent these structures using the type *Node* from figure 6:

In this encoding the message patterns appearing in a a path specification, and matching a set of PDU exchanges, are instances of the free type $\underline{\alpha}$, given by the structure of atomic events as defined by $\mathcal{L}_P$, extended by additional values *MsgParam* for the „leaves“ of the data: Each data field can either be ignored (`ignore`), or a constant single value calculated from the observation state can be required mandatorily (`const`),

or a new binding of a variable can be established for further calculation and condition testing (`read`).

A condition is simply an *Expr* of type `boolean`, which has to evaluate to `true` for the condition to be met.

A state transition is a set of pairs of locations of the observation state and expressions, with all of the former appearing at most once[7].

Starting with „Global Idle“ (i.e. the point to which all traces of the system must return periodically), the set of all possible paths now appears as a *three-colored tree*, cf. figure 5 [8].

3.2 Collecting Constraints for Each Path

The collection of the constraints „ruling“ a given path are rather trivial and mere syntactical transformations. They are realized by the function $[[\_]]^C$, as given in figure 6.

The main purpose of $[[\_]]^C$ is to collect all conditions in the path and conjugate them to one constraint system *cs*. Before this, all identifiers in all expressions will be substituted by *indexed* versions, indicating the step of last assignment to this location, and thus eliminating any ambiguity by the (necessary) reuse of identifiers. For this purpose the step number is calculated in the variable k and and instantiation context κ is maintained, linking each identifier to the step number of the last assignment (either by an `Assign` node or by using it as a `read` parameter of a `Msg`).

Firstly, all expressions are instantiated by replacing each variable reference by the indexed version, depending on this step number of last assignment[9].

Then the constraint system *cs* and the *InstContext* κ are updated, depending on the type of *Node* :

`Test` The (instantiated) expression is simply added to the constraint system.

`Assign` All single assignments in this node are changed to equalities, which are added to the C.S., and κ is updated to the current step number for all identifiers a value is assigned to.

`Msg` This is the most complicated case. Each binding of the data positions of the selected free type is processed independently:

- `ignore` bindings are simply — ignored.
- `const`(e) bindings are replaced by the `read` of a freshly introduced temporary variable J_k, and the equality $J_k = e$ is added to the constraint system *cs*.
- `read`(i) bindings just update the instantiation context.

[7] It is easily seen, that all aspects of $M \diamond R$ from the semantics of $\mathcal{L}_S^2$ can be compiled into sequences of *Node*s: The partiality of the Relation R from the semantic model is explicitely coded as `Test` nodes; the value of the transformation relation is coded as set of assignments to locations of the observation state, and each message set is coded by an `Msg` node, containing bindings from critical message parameter values to local variables, and followed by appropriate `Test` nodes containing predicates on these local variables.

[8] Please node that in figure 6. each *Path* is looked at from leaf to root (as a „Co-Tree“), i.e. it is given as a pair of a *Node* and the *preceding* path (or `top` in case of the root of the tree).

[9] Each *Node* has to be modified accordingly, i.e. all identifiers have to be replaced by the correctly indexed version. For sake of readability this trivial step is left out in the formulae of figure 6.

$$
\begin{array}{lll}
\mathit{Node} & ::= & \texttt{Assign}\ \langle\!\langle\ \mathbb{P}\ (\mathit{Location}\ \times\ \mathit{Expr})\ \rangle\!\rangle \\
& \mid & \texttt{Test}\ \langle\!\langle \mathit{Pred} \rangle\!\rangle \\
& \mid & \texttt{Msg}\ \langle\!\langle \mathit{MsgTags}\ \times\ \texttt{seq}\ \mathit{MsgParam} \rangle\!\rangle \\
\\
\mathit{MsgParam} & ::= & \texttt{read}\ \langle\!\langle \mathit{Location} \rangle\!\rangle \\
& \mid & \texttt{const}\ \langle\!\langle \mathit{Expr} \rangle\!\rangle \\
& \mid & \texttt{ignore}
\end{array}
\qquad
\begin{array}{l}
\mathit{InstContext} == \mathit{Ident} \rightarrow \mathbb{Z} \\
\mathit{IndexedIndent} == \mathit{Ident} \times \mathbb{N}
\end{array}
$$

$$
\begin{array}{l}
\mathit{Path} == \mathit{Node}\ \times\ (\mathit{Path} \cup \{\texttt{top}\}\ ) \\
\mathit{Expr} == \mathcal{L}_F[\{\}][\mathit{Location}]\ .E \\
\mathit{Pred} == \mathcal{L}_F[\{\}][\mathit{Location}]\ .P \\
\mathit{inst}\_\ \_ : \mathit{InstContext} \times \mathit{Expr}[\mathit{Ident}] \nrightarrow \mathit{Expr}[\mathit{IndexedIdent}] \\
[\![\_]\!]^C\ : \mathit{Path}\ \rightarrow\ \mathit{CS}\ \times \mathbb{N} \times \mathit{InstContext} \times \mathit{Path}
\end{array}
$$

$$
\frac{\begin{array}{c}[\![\ b\ ]\!]^C\ = (cs, k, \kappa, b') \\ e'\ =\ \mathit{inst}_\kappa e\end{array}}{[\![\ (\ \texttt{Test}(e),\ b\ )\ ]\!]^C\ =\ (c \cup e', k+1, \kappa, (\texttt{Test}(e'),\ b'\ )\ )}
$$

$$
\frac{\begin{array}{c}[\![\ b\ ]\!]^C\ = (cs, k, \kappa, b') \\ \mathit{collect}^A(x, cs, k, \kappa)\ =\ (cs', \kappa') \\ \mathit{collect}^M(y, cs, k, \kappa)\ =\ (cs'', \kappa'')\end{array}}{\begin{array}{c}[\![\ (\ \texttt{Assign}(x)\ ,\ b\ )\ ]\!]^C\ =\ (cs', k+1, \kappa', b') \\ [\![\ (\ \texttt{Msg}(t, y)\ ,\ b\ )\ ]\!]^C\ =\ (cs'', k+1, \kappa'', b')\end{array}}
$$

$$
\frac{\begin{array}{c}\mathit{collect}^A(x, cs, k, \kappa)\ =\ (cs', \kappa') \\ e'\ =\ \mathit{inst}_\kappa e\end{array}}{\begin{array}{c}\mathit{collect}^A(\{i \mapsto e\} \cup x, cs, k, \kappa) \\ =\ (cs'\ \cup\ \{i_k\ \boxed{=}\ e'\}, \kappa'\ \oplus\ (i \mapsto k))\end{array}}
$$

$$
\frac{\begin{array}{c}\mathit{collect}^M(x, cs, k, \kappa)\ =\ (cs', \kappa') \\ e'\ =\ \mathit{inst}_\kappa e \\ \kappa'\ i\ <\ k\end{array}}{\begin{array}{c}\mathit{collect}^M(\langle \texttt{ignore} \rangle \frown x, cs, k, \kappa)\ =\ (cs', \kappa') \\ \mathit{collect}^M(\langle \texttt{const}\ e \rangle \frown x, cs, k, \kappa)\ =\ (cs'\ \cup \{J_n\ \boxed{=}\ e'\}, \kappa') \\ \mathit{collect}^M(\langle \texttt{read}\ i \rangle \frown x, cs, k, \kappa)\ =\ (cs', \kappa' \oplus (i \mapsto k))\end{array}}
$$

$$
\frac{}{\begin{array}{c}\mathit{collect}^A(\{\}, cs, k, \kappa)\ =\ (cs, \kappa) \\ \mathit{collect}^M(\{\}, cs, k, \kappa)\ =\ (cs, \kappa)\end{array}}
$$

$$
\frac{\begin{array}{c}\mathit{collect}^A(x, cs, k, \kappa)\ =\ (cs', \kappa') \\ \mathit{collect}^M(x, cs, k, \kappa)\ =\ (cs'', \kappa'') \\ \kappa'\ i\ =\ k\end{array}}{\begin{array}{c}\mathit{collect}^A(\{i \mapsto \_\} \cup x, \_, k, \kappa)\ = \\ \mathit{collect}^M(\langle \texttt{read}i \rangle \frown y, \_, k, \kappa)\ = \\ \texttt{Error}(\text{"Duplicate Assignment to Location } i\text{"})\end{array}}
$$

$$
\frac{\kappa\, i\ =\ n\ \ \wedge\ \ n \geq 0}{\mathit{inst}_\kappa\ i\ =\ i_n}
\qquad
\frac{\kappa\, i\ =\ -1}{\mathit{inst}_\kappa\ i\ =\ \texttt{Error}}
$$

$$
\frac{}{\mathit{inst}_\kappa\ \mathit{op}(e_1, \ldots, e_n)\ =\ \mathit{op}(\mathit{inst}_\kappa(e_1), \ldots, \mathit{inst}_\kappa(e_n))}
$$

Figure6. Collecting Constraints for a given Path

The formulae of figure 6 additionally do some error detection, e.g. duplicate use of identifiers in the same transition step or reference to un-initialized variables.

3.3 Interpreting the Constraint System

When performing conformance tests w.r.t. the given Use Case specification, the set of participating actors will be divided into those forming the „System Under Test“, and the others, simulated by the „test harness“.

$$
\begin{array}{l}
A)\ C_n \wedge \bigwedge_{m \neq n} \bullet \neg\ C_m \\
B)\ \exists m, n \bullet m \neq n\ \wedge\ [\![C_m \wedge C_n]\!] \neq \{\}\ \wedge\ D_m = D_n \\
C)\ \exists m, n \bullet m \neq n\ \wedge\ [\![C_m \wedge C_n]\!] \neq \{\}\ \wedge\ D_m = D_n \\
D)\ \bigwedge \neg\ C_n \\
E)\ \bigwedge \neg\ D_n
\end{array}
$$

Figure7. Some questions to the constraint system

The constraint system reflects all conditions which necessarily have to be met for a given path to be taken. Let $N_1 \ldots N_n$ be the set of paths through the specification. Let $C_n \wedge D_n$ be the constraint implied by N_n, and C_n be the part of the constraint, which is *controllable* by the testbed, i.e. is calculated *exclusively* from values contained in messages emitted by the testbed. Assume that $C_n \wedge D_n$ has been transformed to maximize C_n, so that D_n contains only those constraints which are *not* reducible to a controlled value. Now several questions can be asked to this „cleaned-up“ constraint system, cf. figure 7:

A) If this C.S. has a solution, then emitting the corresponding stimuli will force the test to run into branch N_n (or fail).
B) If this property is true, then there are non-deterministic decisions in the specification w.r.t. the testbed behavior.
C) It the C.S. does depend irreducibly on message parameters emitted by the SUT, then coverage of all paths cannot be guaranteed by the test configuration, but has to be monitored during test execution, — with no guarantee of ever happening.
D) If there are „user data“ combinations, which are *not* entailed by some path, then not all possible user inputs are covered in the specification.
E) . . . idem for SUT-generated messages.

3.4 Future Work

Further research is necessary to integrate some features. The most important of these seem to be:

– The temporal restrictions should also be definable by expressions, and these should be integrated into the constraint collection mechanism.
– „Local Loops“, i.e. loops or recursive calls embedded into a linear path would be of large value in practical application (c.f. the „*“ construct used in 9). Can be treated with data-flow analysis or „spectral analysis“ of the updates.
– The „initial state“ has to be modeled and decided. Several strategies seem possible, — further research is required.
– The „interleave“ operator has not yet been integrated.

Acknowledgments

The author wants to thank WOLFGANG GRIESKAMP, microsoft research, who initiated our activities in the field of Use Cases, all colleagues at daimlerChrysler/FT3/SM and at their costumers, who willingly supported the analyzing phase, and the colleagues at TU/ÜBB for critical discussions.

References

[Ber98] Edward V. Berard. Be careful with "use cases". Technical report, The Object Agency, Inc., 1998. `http://www.toa.com/pub/use_cases.htm`.

[BG98] Robert Büssow and Wolfgang Grieskamp. *The ZETA System Documentation*. Technische Universität Berlin, December 1998. URL: `http://uebb.cs.tu-berlin.de/zeta`.

[BG99] Robert Büssow and Wolfgang Grieskamp. A Modular Framework for the Integration of Heterogenous Notations and Tools. In Keijiro Araki, Andy Galloway, and Kenji Taguchi, editors, *Proc. of the 1st Intl. Conference on Integrated Formal Methods – IFM'99*. Springer-Verlag, London, June 1999.

[BGK97] Greg Butler, Peter Grogono, and Ferhat Khende. A Z specification of use cases. In *Proc. of the Asia-Pacific Software Engineering Conference and International Computer Science Conference*, pages 505–506. IEEE Computer Society Press, 1997.

[BN92] S. M. Brien and J. E. Nicholls. Z base standard. Technical Monograph PRG-107, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, UK, November 1992. Accepted for standardization under ISO/IEC JTC1/SC22.

[Col98] Derek Coleman. A use case template: draft for discussion, 1998. Hewlett-Packard Software Initiative.

[GL00] Wolfgang Grieskamp and Markus Lepper. Using Use Cases in Executable Z. In *ICFEM 2000*, September 2000.

[GLST01] Wolfgang Grieskamp, Markus Lepper, Wolfram Schulte, and Nicolai Tillmann. Testable use cases in the abstract state machine language. In *Proc. of the APAQS 2001 conference*. IEEE Computer Society Press, 2001.

[Gri99] Wolfgang Grieskamp. *A Set-Based Calculus and its Implementation*. PhD thesis, Technische Universität Berlin, 1999.

[Gri00] Wolfgang Grieskamp. A Computation Model for Z based on Concurrent Constraint Resolution. To appear in ZB2000 – International Conference of Z and B Users, September 2000.

[Hof01] P. Hofstedt. *Cooperation and Coordination of Constraint Solvers*. PhD thesis, Dresden University of Technology, 2001.

[PCDG00] Peter Pepper, Michael Cebulla, Klaus Didrich, and Wolfgang Grieskamp. From program languages to software languages. *Journal of Systems and Software*, 2000.

[Sei01] Dirk Seifert. Automatische Testfallerzeugung für reaktive Systeme —state of the art. Technical Report 2001-16, Technische Universität Berlin, 2001.

[Ste01] Perdita Stevens. On use cases and their relationships in the Unified Modelling Language. In *Proc. Fundamental Approaches to Software Engineering*, number 2029 in LNCS, pages 140–155. Springer-Verlag, April 2001.

A Further Aspects of Seamless Integration of Well-Founded Languages into Existing Semi-formal Documents in Industrial Practice

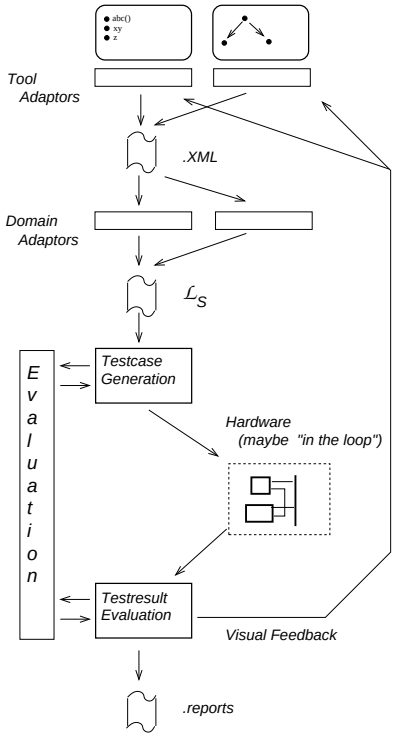

Figure8. The useCASE pipeline

As mentioned above, not only mathematical and engineering, but also ergonomic (or even „pedagogic“) aspects have to be considered when trying to introduce „formal methods“ into industrial practice. Here we present some single aspects and solutions, as proposed in the useCASE project.

A.1 The useCASE Architecture

A central requirement from industrial processes is that new tools and methods can easily be integrated into existing procedures and habits. This requires a certain versatility and adaptability w.r.t. different front-end representations, which in turn requires a modular design of the whole processing pipeline. This modular design of the useCASE-software is depicted in figure 8.

The role of the **front-end** layer can be fulfilled by most diverting editing tools: Either graphical or text based or arbitrarily combined. Proprietary tools or general purpose commercial tools can be used, as long as they offer the necessary import/export facilities.

The **middle-end** layer has to be programmed and consists of two independent and freely combinable layers: The firsts task is to translate the front-end representation format into a normalized intermediate representation of the specification and data definition languages ($\mathcal{L}_S^x$ and $\mathcal{L}_P^x$, as presented in section 2.3) —and to re-import all diagnosis and locator information back into the front-end tool. This layer is called „tool-adaptor“ in figure 8, since it varies with the chosen editing tools, while the languages($\mathcal{L}_S^x$ and $\mathcal{L}_P^x$) are kept constant within a given project.

The second middle-end layer is called „domain-adaptor“. Its job is to map the specification and data format definition languages to the underlying execution model. The mediating data format between these both layers is an XML based encoding, thus permitting the free combination of the same front-end tool with different specification languages, and of different front-end representations for the same specification language.

The **back-end**, which has to do test-data evaluation, analysis and generation, are implemented either by mapping the specification structures to some off-the-shelf constraint evaluating tool (like it had been done with ZAM/Zeta in [GL00]), or has to be implemented from scratch. Even the last version should be not *too* expensive, because w.r.t. the industrial practice the specification languages do not need to support high-order functions, as Zeta did.

A.2 Aspect: Front-end Representation

In the ZeTA based projects a small „use case language“ was realized totally embedded into the Z language: The operators of this language were defined as Z functions on schema data, conveniently writable because being defined as „user defined operators“, a feature gracefully supported by the Zeta compiler.

While everything worked fine on the execution level, and at the same time on the „output side“ all typesetting of the specification formulae looked perfectly professional for a mathematician, — from the the „ergonomic“ view of practical typing, all data and all operators had to be de-notated as LaTeX commands, in different LaTeX environments. This turned out to be a great problem for acceptance in industrial practice.

Therefore our basic approach for all front-end considerations implies, that for each „front-end element“ there is a multitude of different lexical representations, which can be used interchangeable, freely selected depending on the tools the engineer is used to, and which could be used even in a mixed fashion.

The basic approach from the Z standard [BN92] seems canonical and needs only slight modification. For each „token“ or, if necessary, for syntactic constructs there is at least one representation each in the formats...

- pure „Ascii“ text[10],
- a unicode character,
- an XML entity or element definition,
- a LaTeX command.

Usecase *Geldabheben*

- ATM::Aufforderung (KarteEinschieben)
- KD::KarteEinschieben
- ATM::Aufforderung (`Geheimzahl`)
- □
  - 0..20 : KD::EingabePIN (?*num*)
    - ATM::CheckPIN (*num*)
      - □
        - SEC::PINChecked (`OK`)
          - ATM::Aufforderung(`BetragEingeben`)
          - KD::Betragseingabe(?*amount*)
          - Auszahlung(*amount*)
          - ATM::KarteAusgeben
        - SEC::PINChecked (`failed`)
          - ATM::KarteAusgeben
  - 15..17.5 : ATM::KarteAusgeben

Figure9. A simple example with editor controled item lists (e.g. ms-Word “formats”)

A.3 Implementation as an ms-word plug-in

A basic strategy to increase acceptance and performance is to map syntactic constructs of a specification language $\mathcal{L}_S^{\square}$ to so called „format“ objects in the „microsoft Word“ tool: One list format can be used to represent e.g. the concatenation operation, another list format to represent alternatives.

So the writer of the specification can carry on using the type setting mechanisms well-known to her/him, but indeed does write sentences of an exactly defined language. Figure 9 shows an example, in which different ms-word list formats are re-interpreted as constructors of the language $\mathcal{L}_S^2$: • means Sequentialization and □ means alternatives[11].

A.4 Aspect: Smooth Migration by Mixing Informal and Formal Parts of Text

Smooth migration from informal to formal documents can further be supported by an adequate design of the specification language $\mathcal{L}_S^{\square}$: This can be designed that the existing informal interaction descriptions are recognized as comments or as ε-events. So the existing documents as found in an established industrial project are valid, but „empty“, specification documents, and can be enriched step-by-step with formal elements, either replacing the lines-of-text with informal content, or just additionally.

Since the intermediate (or permanently remaining) informal elements have to be treated as ε-transitions, the semantic layers of the specification language have to be able to deal with the well-known complications introduced by those. These informal elements can either be eliminated step-by-step, or may be kept in the documents up to the end of the project as a kind of comment or documentation.

[10] This is what is called „email-format“ in the Z standard.

[11] Please note that it may be in-house style to use the local language instead of English.