Preface

These are the proceedings of the Visual Formal Methods – VFM'99 symposium, held on August 23rd 1999 in Eindhoven, The Netherlands.

The idea behind Formal Methods has always been to have methods allowing for unambiguous, mathematically precise descriptions of a system, supporting formal analysis. However, the appearance of a Formal Method is often based on its underlying mathematical theory, which, although intuitive to the people specialised in this particular Formal Method, is not particularly intuitive to outsiders. Although the use of Formal Methods in the industry has been prophesied since their very dawn, reality learns that their acceptance is still not very wide-spread.

On the other hand, an increasing interest is showing nowadays directed towards visual languages. A reason for this trend is the fact that visual languages are much closer to the way in which people think when developing a system; their appearance is intuitive and they allow for expressing the relation between objects spatially, thus easing the understanding of descriptions.

Combining the strengths of both approaches will result in a formal and still intuitive approach to system design. Examples of such combinations already exist; visual languages with a formal semantics which allows for formal validation (e.g. Message Sequence Charts) and Formal Methods with a graphical interface. It looks as if in the last few years, the research community is taking up this challenge, and the combined use of Formal Methods and Visual Languages is becoming a topic of interest.

Taking up on this trend, it was decided in the group of Formal Methods at the department of Computing Science at the Eindhoven University of Technology, that a symposium on this topic should be organised. This symposium, christened Visual Formal Methods – VFM'99, linked to Concur'99 aimed at bringing together leading researchers and practitioners from the intersecting area of Visual Languages and Formal Methods.

The area of Visual Formal Methods is rather broad but roughly it can be divided into two distinct areas. On the one hand, methods are being developed to deal with the architecture of systems. Often, the architecture, with the emphasis on software architecture, plays an important role in the development of software systems. Various methods for coping with the problems involved in systems' architecture are investigated, each emphasising different aspects of this area as is shown in "Formalization and Visualization of Software Architectures", which suggests how the module view of software architecture can be visualised and formalised. The presentation on "Object-Oriented Modelling and Specification using SHE" discusses a system level modelling language, POOSL, which is used to manage the intrinsic complexity of complex hardware and software systems. The use of a visual formal method is not limited to software architecture as is convincingly demonstrated in "Visualising business processes", which describes how the formal visual language AMBER is applied to business architectures.

On the other hand, research is directed towards the specification and analysis of the behaviour of complex software and hardware systems. First of all, languages itself are the subject of investigation, as is shown in "Composing Automata in Graphical Languages for Reactive Systems: from Argos to Mode-Automata". There, a graphical, synchronous approach to specifying reactive systems is explained. Secondly, development of tool support for the analysis and verification of systems is an important research area. In "Visual Temporal Logics as a Rapid Prototyping Tool" formal specifications in real-time symbolic timing diagrams are used for reliable and rapid prototyping. The presentations on "Automatic Synthesis of SDL from MSC in Forward and Reverse Engineering" and "uBET: Graphical Specification Support in an Industrial Environment" focus on the use of computer-aided software tools and the effects they have on the time-to-market.

We thank Desiree Meijers, Marcella de Rooij and Anne-Meta Oversteegen for their organisational assistance. For providing and setting up the required hardware and software support, BCF is thanked. Last but not least, we are grateful for the financial support by SOBU (the Co-operation Center of Tilburg and Eindhoven Universities) IPA (the Institute of Programming Research and Algorithmics, NL), and the Eindhoven University of Technology.

Dragan Bošnački	August 1999
Sjouke Mauw	Eindhoven
Tim Willemse	The Netherlands

Table of contents

Visual Temporal Logic as a Rapid Prototyping Tool[⋆]

Martin Fränzle and Karsten Lüth

Carl von Ossietzky Universität Oldenburg
Department of Computer Science
P.O. Box 2503, D-26111 Oldenburg, Germany
{Martin.Fraenzle|Karsten.Lueth}@Informatik.Uni-Oldenburg.De

Abstract. As embedded systems become more and more complex, early availability of unambiguous specification of their intended behaviour has become an important factor for quality and timely delivery. Consequently, the quest for rapid prototyping methods for such specifications arises. Addressing these issues, the computer architecture group of Oldenburg University has devoted a major line of research towards automatic prototyping of embedded controllers from fully formal specifications given as real-time symbolic timing diagrams (RTSTDs, for short). RTSTDs are a graphical formalism for specifying behavioural requirements on hard real-time embedded systems. They are a full-fledged metric-time temporal logic, but with a graphical syntax reminiscent of the informal timing diagrams widely used in electrical engineering.
Within this survey article, we will explain real-time symbolic timing diagrams as well as the ICOS tool-box supporting RTSTD-based requirements capture and rapid prototyping. ICOS integrates a variety of tools for RTSTDs, ranging from graphical specification editors over tautology checking and counterexample generation to code generators emitting C or VHDL, thus bridging the gap from specification to prototype generation.

1 Introduction

Due to rapidly dropping cost and the increasing power and flexibility of embedded digital hardware, digital control is becoming ubiquitous in technical systems encountered in everyday life. Often, such embedded systems can hardly be altered once they have been shipped out due to extremely large quantities being shipped, or they even have to be right first time due to their crucial impact on safety of human life. For example, modern means of transport rely on digital hardware even in vital sub-systems like anti-locking brakes, fly-by-wire systems, or signaling hardware, as does medical equipment even in such critical applications as life-support systems or radiation treatment. As, furthermore, many

[⋆] This article reflects work that has been partially funded by the German Research Council DFG under grant no. Da 205/5-1. A preliminary version of this article has been published as [6].

embedded systems are developed under tight time-to-market constraints, early availability of unambiguous specification of their intended behaviour has become an important factor for quality and timely delivery.

Such a specification is, however, always a compromise between various demands: ought it to be operational, in order to guide developers and programmers in the implementation phase, or ought it to be declarative in order to allow straightforward formulation of safety requirements, thus supporting safety analysis? Ought it to be formal, thus facilitating formal analysis and hence providing correctness guarantees that are not otherwise available, or may it be informal if this enhances comprehensibility, thus simplifying traditional approaches for ensuring quality of software, like testing and certification by code inspection?

Due to their prospects for reconciling some of these seemingly contradictory demands, graphical specifications have recently attracted much interest. Their prospects for gaining comprehensibility of even complex specification are deemed so high that it should be possible to simultaneously further the level of formality without sacrificing understandability. However, even with graphical specifications, the global interaction patterns of complex (for example, distributed) systems remain complex and, furthermore, it seems that graphical idioms also tend to hide some of the fine-grain semantics from the user, at least from the non-expert one. Thus, while being a big step ahead, graphical specification formalisms are not per se a means for ensuring that "what you specify is what you mean".

The problem of misconceptions in early development phases is, however, well-known in software engineering. A traditional remedy is *rapid prototyping*, where a partially developed product is brought into executable shape in order to assess its compliance with expectations. We suggest to take over the paradigm of "rapid prototyping" to the realm of formal, graphical, and declarative (i.e., essentially non-operational) specifications such that these become executable, thereby facilitating early evaluation of specifications on an operational model. If such a prototyping process is based on an unambiguous semantics of specifications and applies rigorous rules for deriving executables from specifications then it can, furthermore, be made sure that the prototype obtained is in strict correspondence with the specification such that the evaluation is faithful.

In this article, we propose a method of that kind: it builds upon a fully formal semantics of specifications and applies automata-theoretic constructions for fully automatically deriving operational systems that represent the specification in a "what you specify is what you get" way. The method has been developed and implemented by the computer architecture group of Oldenburg University, which has dedicated part of its rapid prototyping project 'EVENTS' [13] towards automatic prototyping of embedded control hardware from fully formal specifications given as *real-time symbolic timing diagrams*. Real-time symbolic timing diagrams (RTSTDs, for short), as introduced in [4], are a graphical formalism for specifying behavioural requirements on hard real-time embedded systems. They are a full-fledged *metric-time temporal logic*, but with a graphical syntax reminiscent of the informal timing diagrams widely used in electrical engineering.

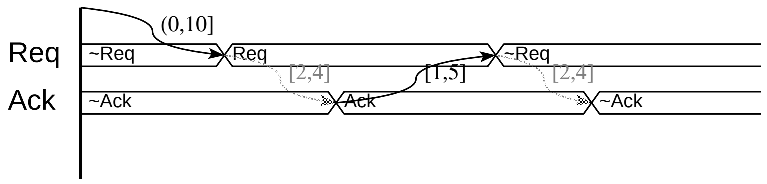


Fig. 1. An RTSTD specifying a simple handshaking protocol. The black arcs represent strong constraints, while weak constraints (i.e., assumptions on the environment) are printed in grey. The perpendicular line to the far left indicates the activation mode of the diagram, which here is the so-called invariant mode, meaning that the diagram has to apply whenever its activation condition fires. The activation condition is the state condition crossed by the perpendicular line, i.e. $\neg\texttt{Req} \wedge \neg\texttt{Ack}$.

In Sect. 2 and 3, we introduce real-time symbolic timing diagrams and give an overview over the ICOS tool-box supporting requirements capture and rapid prototyping using RTSTDs. As the ICOS approach to rapid prototyping is based on synthesis of embedded control hardware — in general, FPGAs — satisfying the specification, we then turn to game-theoretic methods of controller synthesis. Section 4 introduces the controller synthesis problem, and Sect. 4.1 outlines a classical controller synthesis framework based on the effective correspondence of propositional temporal logic to finite automata on infinite words and on the theory of ω-regular games [18]. A compositional variant of this approach, which is more suitable for rapid prototyping purposes due to its reduced complexity, is shown in section 4.2. This is our current synthesis method, which has been fully implemented in the ICOS tools [10, 12]. The results obtained using this method on e.g. the FZI production cell [11] indicate that the compositional extension yields a significant enhancement for reactive systems, yet a further compositional treatment of timing is necessary for real-time systems. Section 4.3 sketches the basic design decisions underlying such an extension which is currently being implemented for a new release of ICOS, while Sect. 5 compares this to the state of the art.

2 Real-time symbolic timing diagrams

The RTSTD language is a metric discrete-time temporal logic with an — as we hope — intuitive graphical syntax reminiscent of the informal timing diagrams widely used in electrical engineering, and with a declarative semantics which is formalized through a mapping to propositional temporal logic (PTL). In contrast to some other approaches using timing diagrams, e.g. those described in [1, 9], symbolic timing diagrams do *not* have any imperative control structure like iteration or sequencing. Instead, the recurrence structure of RTSTDs is expressed in terms of the modalities of linear-time temporal logic, thus providing a direct logical interpretation. In fact, RTSTDs provide a declarative rather than an operational specification style, even despite their intuitive syntax: an RTSTD is interpreted as a constraint on the admissible behaviours of a component, and a typical specification consists of several small RTSTDs, with the individual constraints being conjoined. The main consequence is that RTSTDs are well-suited for incremental development of requirements specifications. However, they pose harder problems than more operational timing diagrams when used as source language for code generation. Fig. 1 shows an example RTSTD, specifying a simple handshaking protocol using two signals `Ack` and `Req`.

A basic[1] RTSTD consists of the following parts:

– An *entity declaration* defining the interface of the component (not shown in Fig. 1). It specifies the signals (e.g. `Req, Ack`), their data types (e.g. `Bit`) and access modes (`in` or `out`, i.e. being driven by the environment or the component, resp.).
– A set of *waveforms* (here `Req,Ack`). A waveform defines a sequence of Boolean expressions associated with a signal (e.g. ~`Req`, then `Req`, then ~`Req` in the upper waveform of Fig. 1). The point of change from validity of one expression to another is called an *event*. A distinguished *activation event*, which specifies a precondition for applicability of the diagram, is located to the left over the waveforms.
– A set of *constraints*, denoted by constraint arcs, which define a partial order on events. We distinguish between *strong constraints* and *weak constraints*. Strong constraints are those which have to be satisfied by the system under development, while weak constraints denote assumptions on the behaviour of the environment. Violation of a weak constraint implies that the remainder of the timing diagram poses no further design obligations. Table 1 summarizes the different kinds of constraints.
– An *activation mode*. *Initial diagrams* describe requirements on the initial system states whereas an *invariant diagrams* expresses requirements which must be satisfied at any time during system lifetime. Invariant mode corresponds to the ‘always’ modality of linear-time temporal logic, implying that — in contrast to timing diagram formalisms employing iteration — multiple incarnations of the diagram body may be simultaneously active if the activation condition fires again before the end of the body has been reached.

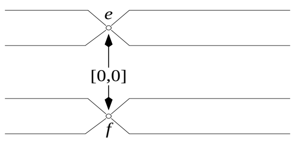
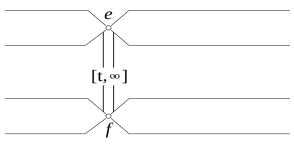
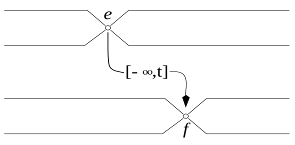
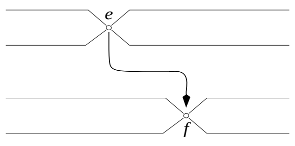
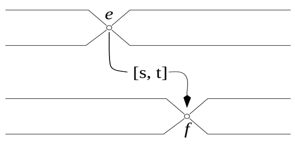
More details about syntax and semantics of RTSTDs are given in [4].

3 The ICOS tool-box

Despite the appealing graphical syntax of RTSTDs, specification of reactive systems using RTSTDs remains a challenging task. The ICOS tool-box, as shown in Fig. 2, is a comprehensive set of tools that supports this process [5, 10, 12]. ICOS is built around a timing diagram database that holds all specification clauses belonging to the current project. The database is augmented by a layer of procedures for compiling RTSTDs to other effective representations of reactive behaviour, like propositional temporal logic (PTL) [14] or Büchi automata

[1] there is some syntactic sugar available for making large specifications more concise.

Table 1. Basic strong constraint types of symbolic timing diagrams and a compound constraint. Each of these has a weak counterpart, denoted by a shaded constraint arc.

Simultaneity constraint	**Conflict$_t$ constraint**	**Leads-to$_t$ constraint**
		
Events e and f have to occur simultaneously.	e and f do not occur less than t time-units apart.	Event f occurs no later than t time-units after e.
Precedence constraint	**The compound constraint type used in Fig. 1**	
		
Event f does not occur before e.	A conjunction of precedence, conflict$_s$, and leads-to$_t$ constraints.	

[18]. These procedures, as well as the corresponding derived representations of reactive behaviour, are in general hidden from the user. Instead, they are invoked (resp., constructed) on the fly whenever a particular representation is needed for some user-selected task (e.g., for code generation).

User-selectable activities are concerned with design, verification, simulation, synthesis, and debugging. *Design*-related activities involve creating and editing RTSTDs with a graphical editor, as well as browsing the specification database. *Verification* entails static analysis, for checking whether interfaces are consistently used, and automatic tautology checking, for formally verifying that some refined specification satisfies the original commitments of the design task. *Simulation* is performed upon a keystroke by first generating a finite automaton faithfully reflecting the semantics of the specification, then encoding it in the C programming language, and finally linking it to a simulation environment, thus facilitating interactive simulation of specifications. The intermediate steps involved do, however, not require any user interaction.

Simulation shares most of the basic technology with synthesis. *Synthesis* goes however a bit further in that it actually yields an FPGA-based embedded control device that can be plugged into the application context rather than just an interactive simulation. The steps involved are, first, the generation of a set of interacting Moore automata satisfying the specification, second, their translation to synthesizable VHDL code, and third, synthesis of a corresponding FPGA. We will expand on the techniques underlying synthesis in the next section, as embedded controller synthesis forms the backbone of our rapid prototyping method.

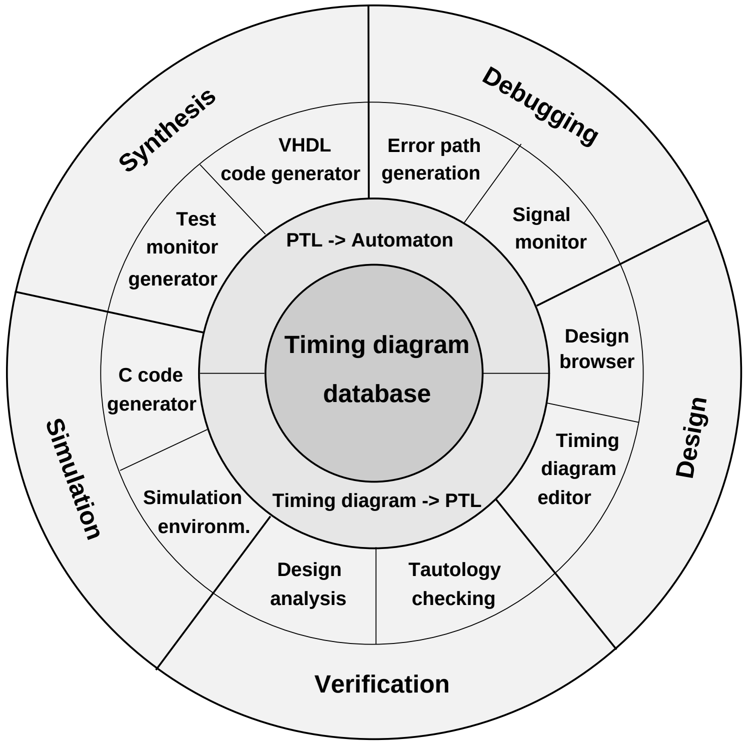


Fig. 2. The ICOS system

Test monitor generation is a minor variant of this synthesis process which yields an FPGA that does not control its application context, but instead may be plugged into the application context as an online monitor, then monitoring the running system for violations of the specification.

Debugging, finally, yields error paths pinpointing the problem whenever synthesis fails due to a contradictory (and thus unimplementable) specification, or if tautology checking finds a non-tautology.

4 Synthesis

Automatic synthesis of FPGA-based controllers is the core of our rapid prototyping method for RTSTD specifications. The application scenario is that the user invokes an *fully automatic* synthesis procedure once the requirement set, being formalised through RTSTDs, is deemed sufficiently complete. The synthesis algorithm then tries to construct an FPGA-based embedded controller satisfying the specification — i.e., an operational prototype — or, if it detects unimplementability of the specification — delivers diagnostic information.

In fact, the synthesis method is a three-step procedure: first, the synthesis algorithm tries to construct a Moore automaton satisfying all stated requirements. However, no appropriate Moore automaton need exist due to a contradictory specification in which case the algorithm creates error paths to help the programmer refine the specification. Otherwise, the second step can commence, where the generated Moore automaton is automatically translated to VHDL code. The subset of VHDL used as target language is synthesizable by the Synopsys tools [7] such that the final step of actually implementing the automaton by a given target technology (e.g. FPGAs) can be done through so-called high-level synthesis by the Synopsys tools [7], and we have indeed integrated our tools with the Synopsys and some Xilinx tools to achieve this.

As high-level synthesis is by now an industrially available technology, we will in the remainder of this article concentrate on the first step and will sketch different specification-level synthesis procedures yielding sets of interacting Moore machines from RTSTD specifications. The procedures differ in the methods used for dealing with timing constraints and in the number and shapes of the interacting Moore machines generated, which affects the average-case complexity of the synthesis procedure and the size of the hardware devices delivered.

Given a specification ϕ, the problem of constructing a Moore automaton satisfying the specification, called the *synthesis problem*, is to find a Moore automaton A_ϕ which is

1. *well-typed wrt.* ϕ, i.e. has the inputs and outputs that the entity declaration of ϕ requires, and accepts any input at any time (i.e., only the outputs are constrained),
2. *correct wrt.* ϕ, i.e. the possible behaviours of A_ϕ are allowed by the specification ϕ, formally $\mathcal{L}_{A_\phi} \subseteq \mathcal{M}[\![\phi]\!]$, where $\mathcal{M}[\![\phi]\!]$ is the set of behaviours satisfying ϕ and $\mathcal{L}_{A_\phi}$ is the set of possible behaviours of A_ϕ.

By adoption of logical standard terminology, an algorithm solving the synthesis problem is called *sound* iff, given any specification ϕ, it either delivers an automaton that is well-typed and correct wrt. the specification ϕ, or no automaton at all. It is called *complete* iff it delivers an automaton whenever a well-typed and correct automaton wrt. the specification exists.

4.1 Classical controller synthesis and RTSTDs

If the requirements specification language is a classical, non-graphical, discrete-time temporal logic, like propositional temporal logic (PTL) [14], then algorithms for solving the synthesis problem are well-known: there is an effective mapping of these logics to infinite regular games which, together with the firmly developed theory of strategy construction in infinite regular games yields a fully automatic procedure for generating finite-state winning strategies, i.e. finite-state controllers, from temporal logics specifications of the allowable behaviour [18].

As there is an effective mapping of RTSTDs to PTL, which is fully explored in [10], this approach can be extended to deal with RTSTDs also. Soundness

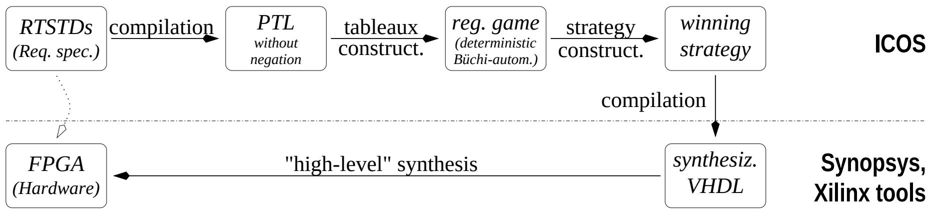


Fig. 3. The basic synthesis chain.

and completeness of the synthesis method then is directly inherited from the corresponding properties of winning strategy construction in regular games. In fact, this chain of algorithms, depicted in Fig. 3, forms the backbone of the ICOS tool set [5, 12].

However, this basic synthesis chain suffers from complexity problems if the specification is large, i.e. is a conjunction of multiple timing diagrams, as the regular games constructed then tend to suffer from state explosion. With the basic method, game graphs grow exponentially in the number of timing diagrams due to the automaton product involved in dealing with conjunction. As this would render application for rapid prototyping impractical, the ICOS tools offer modified procedures which reduce the complexity of dealing with large specifications. Obviously, such extensions cannot deal efficiently with arbitrary RTSTD specifications, but they are, however, carefully designed to cover the typical specification patterns.

4.2 Compositional synthesis

The first such variant is a compositional extension of the basic algorithm. Within this approach, which is sketched in Fig. 4, the specification is first partitioned into a maximal set of groups of formulae $\mathcal{G}_1, ..., \mathcal{G}_n$ such that each output is constrained by the formulae of at most one group. Then synthesis down to a winning strategy is performed for each group individually, yielding for each group $\mathcal{G}_i$ a Moore automaton A_i that has just the outputs constrained by $\mathcal{G}_i$ as outputs, and all other ports as inputs. The individual Moore automata are then compiled to synthesizable VHDL and composed by parallel composition.

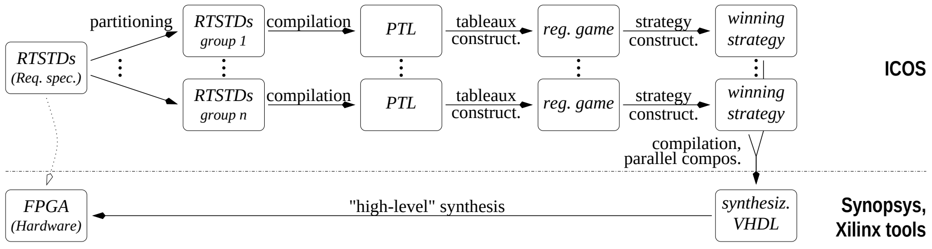


Fig. 4. Compositional synthesis.

With this compositional method, growth of the game graph is linear in the number of groups, and exponential growth is only encountered in the maximal size of the individual groups. Table 2 provides empiric results from [5], obtained by Feyerabend and Schlör when using both the non-compositional and the compositional synthesis procedure of ICOS for synthesizing a controller for the FZI production cell [11]. Overall, Feyerabend and Schlör have found the compositional approach to save over 99% of the transitions in the game graphs of the major components.

Soundness of the compositional synthesis technique is easily established:

Theorem 1. *Let $\mathcal{G}_1, ..., \mathcal{G}_n$ be groups of RTSTDs with O_i, $1 \leq i \leq n$, being the outputs constrained by $\mathcal{G}_i$, and with $O_i \cap O_j = \emptyset$ for $i \neq j$. Let A_i be the Moore automaton synthesized for $\mathcal{G}_i$. Then $A_1 \parallel \ldots \parallel A_n$ is well-typed and correct wrt. $\mathcal{G}_1 \wedge \ldots \wedge \mathcal{G}_n$, where $\parallel$ denotes parallel composition of Moore automata.*

Proof. Correctness of $A_1 \parallel \ldots \parallel A_n$ wrt. $\mathcal{G}_1 \wedge \ldots \wedge \mathcal{G}_n$ is straightforward, as parallel composition of Moore automata with disjoint output alphabets semantically yields language intersection, as does conjunction of RTSTDs. Thus, soundness of the basic synthesis algorithm, which yields $\mathcal{L}_{A_i} \subseteq \mathcal{M}[\![\mathcal{G}_i]\!]$ for the individual groups, implies $\mathcal{L}_{A_1 \parallel \ldots \parallel A_n} = \mathcal{L}_{A_1} \cap \ldots \cap \mathcal{L}_{A_n} \subseteq \mathcal{M}[\![\mathcal{G}_1]\!] \cap \ldots \cap \mathcal{M}[\![\mathcal{G}_n]\!] = \mathcal{M}[\![\mathcal{G}_1 \wedge \ldots \wedge \mathcal{G}_n]\!]$.
Similarly, well-typedness of $A_1 \parallel \ldots \parallel A_n$ wrt. $\mathcal{G}_1 \wedge \ldots \wedge \mathcal{G}_n$ follows from soundness of the basic synthesis procedure since the composition rules for interfaces agree for Moore automata and RTSTDs if outputs occur at most once in a parallel composition. $\square$

Completeness is, however, lost using compositional synthesis. The problem is that a certain group may not be synthesizable without knowledge about the behaviour of another group. Such problems are regularly encountered within compositional methods, and we propose to solve them by just the same techniques that proved to be helpful in compositional verification: the necessary information on the other components can be formalized via assumptions (for example, through weak constraint arcs). It should be noted that ICOS helps in finding adequate assumptions, as an error path is supplied whenever synthesis of a component fails.

4.3 Synthesizing hardware clocks

However, there still is some reason for dissatisfaction: as timing annotations have to be unwound to an according number of next-operators of PTL by the translation of RTSTDs to PTL, which introduces corresponding chains of unit-delay transitions in the game graphs, the modular synthesis method remains exponential in the number of potentially overlapping timing constraints. This makes dealing with timing constraints of more than a handful time units hardly affordable — realistic real-time system programming cannot be done with such code generation methods. Therefore, we are heading for an algorithm that is of linear complexity in the number of timing constraints, even though completeness is thereby necessarily sacrificed. What is thus needed is a synthesis method that separates generation of timers controlling the allowable passage of time from

Table 2. Experimental results obtained when applying non-compositional and compositional synthesis to the Karlsruhe production cell [11] (after [5]).

Non-compositional synthesis: System has 8 components. One controller per component is synthesized. The most complex components are:

Component	STDs	Game graph		Controller		Time (s)
		states	transitions	states	transitions	
Press	14	*1703*	*115897*	82	435	*4483*
Crane	*16*	1574	76041	*190*	*1303*	1035

Compositional synthesis: System is automatically partitioned into 26 groups. One controller per group is synthesized. Most complex group deals with controlling vertical movement of the press:

	STDs	Game graph		Controller		Time (s)
		states	transitions	states	transitions	
(Press, vertical movement)	*5*	*27*	*193*	*6*	*26*	*8*

synthesis of an untimed control skeleton. In the remainder, we sketch a synthesis technique currently being integrated into ICOS, which offers the desired separation. In this hybrid approach, small — and thus harmless wrt. the state explosion problem — time constants are treated by purely ω-automata-theoretic means, whereas large-scale timing constraints, if found to be sufficiently independent, are directly mapped to hardware timers.

The new approach starts by using timed automata for representing the semantics of real-time symbolic timing diagrams: every RTSTD ϕ is assigned a timed automaton A_ϕ that accepts exactly the *counterexamples* of ϕ, i.e. all those traces that are *not* models of ϕ. An example of a timed automaton recognizing the counterexamples of an RTSTD can be seen on the left hand side of Fig. 6, where the conjunction of a precedence and a leads-to$_t$ constraint between events e and f is dealt with.

Once a timed automaton accepting counterexamples to the specification is constructed, synthesis of a controller satisfying the specification can commence: in a first step, all clocks implementing delays of more than a handful time units are removed from the timed automaton and replaced by external timer com-

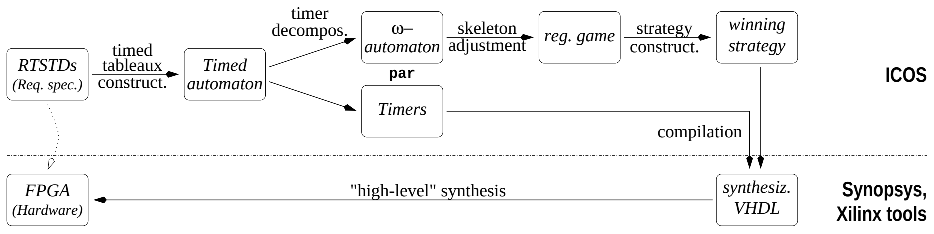


Fig. 5. Synthesis chain employing timer decomposition.

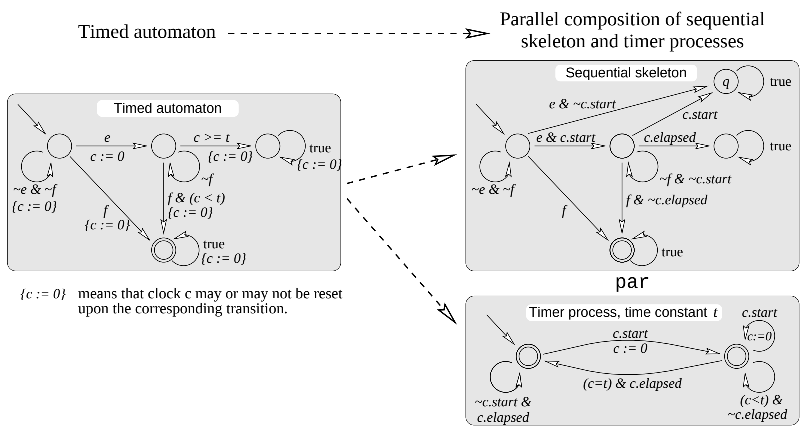


Fig. 6. Converting clocks to timer components.

ponents acting in parallel, as shown in Fig. 6. Thereafter, the remaining clocks are removed by expanding their effect to an appropriate number of unit-delay transitions. This results in an untimed automaton, called sequential skeleton in the remainder, which communicates with the environment and with the timer components. The novelty of our approach is that from then on, synthesis will treat the timers similar to environment components, which means that the behaviour of these components is left untouched during synthesis. The advantages are twofold: first of all, the fixed behavioural description of timers allows for the use of pre-fabricated, optimized hardware components, and second, controller synthesis can concentrate on solving the control problem described by the small, untimed automaton that remains.

Note that the parallel composition (alas automaton product, thus yielding language intersection) of the sequential skeleton and the timer processes derived thus far recognizes the counterexamples to the specification. Now, we would like to implement the timers in hardware and to remove them from the synthesis problem, i.e. we want to synthesize wrt. the skeleton *only* without, however, risking erroneous behaviour of the synthesized controller. As the timer communications are only internal to the controller, the correctness criterion involved is

$$(\mathcal{L}_C \cap \mathcal{L}_{timer}) \setminus [t.start, t.elapsed] \subseteq \overline{(\mathcal{L} \cap \mathcal{L}_t) \setminus [t.start, t.elapsed]}\ , \tag{1}$$

where $\mathcal{L}_C$ and $\mathcal{L}_{timer}$ are the trace sets of the controller and the timers, resp., $\mathcal{L}$ is the language accepted by the skeleton automaton, '$\setminus[t.start, t.elapsed]$' denotes hiding of the timer communications, and the overbar denotes language complement.

It might seem that straightforward synthesis wrt. the complement of the skeleton, which yields a controller C satisfying the language inclusion property $\mathcal{L}_C \subseteq \overline{\mathcal{L}}$, suffices. Unfortunately, $\mathcal{L}_C \subseteq \overline{\mathcal{L}}$ is in general not a sufficient condition for (1) due to the existential quantification involved in hiding,[2] which changes to universal quantification under the complementation involved in (1).

However, this can be repaired by synthesizing wrt. an appropriately adjusted variant *skel′* of the skeleton automaton that enforces a certain usage of timers. The key issue is that the synthesized controller is forced to start a timer and not interfere its run whenever a violation of the corresponding timing constraint could possibly occur. The new skeleton *skel′* is generated by taking the same state set, same initial states, and same transition relation as in the original skeleton, yet expanding the set of accepting states by states like state q in Fig. 6, which is entered if the timer signaling the possible violation of the leads-to$_t$ constraint is not properly activated. The detailed construction, which we cannot provide due to lack of space, is s.t. if a timer action sequence ts exists with $w \oplus ts \in \mathcal{L} \cap \mathcal{L}_{timer}$ then $w \oplus ts' \in \mathcal{L}_{skel'}$ for each $ts' \in \mathcal{L}_{timer}$. Note that furthermore $\mathcal{L}_{skel'} \supseteq \mathcal{L}$ by construction.

If we now synthesize a controller with inputs $I \cup \{t.elapsed\}$, outputs $O \cup \{t.start\}$ (where I and O are the original in- and outputs) that is correct wrt. the adjusted control problem, i.e. satisfies $\mathcal{L}_C \subseteq \overline{\mathcal{L}_{skel'}}$, then we have obtained a correct solution:

Lemma 1. *Correctness of C wrt. $\overline{\mathcal{L}_{skel'}}$, i.e. $\mathcal{L}_C \subseteq \overline{\mathcal{L}_{skel'}}$, implies (1).*

Proof. Assume that $\mathcal{L}_C \subseteq \overline{\mathcal{L}_{skel'}}$ holds and (1) is false, i.e. there exists some $w \in (\mathcal{L}_C \cap \mathcal{L}_{timer}) \setminus [t.start, t.elapsed] \cap (\mathcal{L} \cap \mathcal{L}_{timer}) \setminus [t.start, t.elapsed]$. By definition of hiding this implies that there are two timer action sequences ts_1, ts_2 with

$$w \oplus ts_1 \in \mathcal{L}_C \cap \mathcal{L}_{timer} \quad \wedge \quad w \oplus ts_2 \in \mathcal{L} \cap \mathcal{L}_{timer} \ .$$

Then, by construction of $skel'$, $w \oplus ts_1 \in \mathcal{L}_{skel'}$. But on the other hand $w \oplus ts_1 \in \mathcal{L}_C$ and $\mathcal{L}_C \subseteq \overline{\mathcal{L}_{skel'}}$, which yields a contradiction. □

Consequently, synthesis wrt. the adjusted control problem is sound. Furthermore, by directly mapping all timing constraints of significant size to hardware timers, this method is linear in the number of timing constraints. If it is furthermore combined with modular synthesis then super-linear blow-up can only occur through individual timing diagrams containing unusually large numbers of events or through large groups of formulae controlling the same outputs. These situations are, however, atypical, rendering the new method a practical rapid-prototyping tool for real-time embedded controllers.

Similar methods are, btw., currently explored by the second author for dealing with data in an essentially uninterpreted way. The idea is that thus the traditional separation between control and data can be incorporated into game-theoretic synthesis methods. The net effect is that a large data path no longer yields a blow-up of the game-graph. The complexity of automatic synthesis then predominantly depends on the control part, making automatic synthesis applicable to much larger, control-dominated systems.

[2] For a language $\mathcal{L} \subseteq (\alpha \times (\{t.start, t.elapsed\} \to \mathbb{B}))^\omega$ and some string $w \in \alpha^\omega$, $w \in \mathcal{L} \setminus [t.start, t.elapsed]$ holds if some sequence $ts \in (\{t.start, t.elapsed\} \to \mathbb{B})^\omega$ of timer actions exists with $w \oplus ts \in \mathcal{L}$.

5 Discussion

We have presented a rapid-prototyping framework for requirements specifications that are formalised through real-time symbolic timing diagrams, a metric-time temporal logic with a graphical syntax akin to the informal timing diagrams used in electrical engineering [4]. The underlying core technology is fully automatic synthesis of embedded control hardware from requirements specifications. This involves two major steps: first, the generation of Moore automata satisfying the specification, and second, the actual implementation of these automata by embedded control hardware. For the latter step, we rely on industrially available tools from Synopsys and Xilinx that perform FPGA-based implementation of VHDL-coded Moore-automata [7]. In contrast, the former step is based on procedures that have been specifically implemented for the ICOS tool-box [5, 10, 12]. While the underlying algorithms have been derived from the well-established theory of winning-strategy construction in ω-regular games, the overall approach is — being targeted towards rapid prototyping — mostly pragmatic, weighing efficiency higher than completeness. Two key issues have been identified in this respect: first, the necessity of compositional synthesis of parallel components and second, early decomposition of timing issues from the synthesis problem. The result is a synthesis method that is essentially linear in the size of the specification and thus suitable as a development tool in rapid prototyping contexts.

Within the ICOS tool-box, these algorithms are closely integrated with a graphical specification editor supporting the specification phase and all compilation and mapping algorithms necessary for mechanically translating down to automatically synthesizable VHDL code such that implementation is truly automatic from the specification level all the way down to actual hardware. ICOS furthermore comprises — under a common user interface — tools for browsing and manipulating a specification database, as well as for interactively simulating and for verifying specifications. It is thus a comprehensive environment for the incremental development of RTSTD-based requirements specifications. Furthermore, within the rapid-prototyping project "EVENTS" [13], ICOS is conjoined with Statemate-based code generation techniques such that rapid prototyping of hybridly specified systems, where some components have a declarative formulation through RTSTDs while others are operationally described by Statecharts [8], becomes feasible. While this involves integration of different rapid-prototyping tools through automatic interface generation for the generated components, other extensions to the source language accepted can be accomplished within just the ICOS tool: ICOS is modularly built such that while the current version is dedicated towards RTSTD-based specification, an adaptation to other declarative specification formalisms is possible. A possible candidate formalism, for which front-end tools like graphical editors are already under development at our department, is that of Live Sequence Charts (LSCs), an extension of Message Sequence Charts (MSCs) recently proposed by Damm and Harel [3].

On the algorithmic side, we think that the main contribution of the ICOS tools to game-theoretic synthesis are methods for dealing with timing or with the data path in an uninterpreted way. Due to the obvious necessity of treating timing mostly independent from algorithmic aspects within any reasonably efficient synthesis method for hard real-time controllers, quite a few other research groups work on this theme. However, most approaches are based on primarily operational rather than declarative specification styles (e.g. [19]). Closest to our approach is [15], where Dierks and Olderog detail a direct mechanism for deriving timer actions from specifications formalised through a very restrictive subset of Duration Calculus [20], the so-called DC-implementables [16]. Dierks' algorithm is extremely efficient, but at the price of a very restrictive specification format: the processable specifications are confined to be 'phased designs' in the sense of the ProCoS projects [2], which are akin to RTSTDs featuring just three events. While our formalism is more expressive in this respect, Dierks and Olderog do, on the other hand, go ahead by dealing with a dense-time logic and analyzing certain kinds of switching latency.

References

1. G. Borriello. Formalized timing diagrams. In *The European Conference on Design Automation*, pages 372–377, Brussels, Belgium, Mar. 1992. IEEE Computer Society Press.
2. J. P. Bowen, M. Fränzle, E.-R. Olderog, and A. P. Ravn. Developing correct systems. In *Proc. 5th Euromicro Workshop on Real-Time Systems, Oulu, Finland*, pages 176–189. IEEE Computer Society Press, June 1993.
3. W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. Technical Report CS98-09, Weizmann Institute, Apr. 1998.
4. K. Feyerabend and B. Josko. A visual formalism for real time requirement specification. In *Transformation-Based Reactive System Development*, number 1231 in LNCS, pages 156–168. Springer Verlag, 1997.
5. K. Feyerabend and R. Schlör. Hardware synthesis from requirement specifications. In *Proceedings of EURO-DAC'96 with EURO-VHDL'96*. IEEE Computer Society Press, September 1996.
6. M. Fränzle and K. Lüth. Compiling graphical real-time specifications into silicon. In Ravn and Rischel [17], pages 272–281.
7. S. Golsen. State Machine Design Techniques for Verilog and VHDL. *Synopsys Journal of High Level Design*, Sept. 1994.
8. D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and A. Shtul-Trauring. STATEMATE; a working environment for the development of complex reactive systems. In *Proceedings of the 10th International Conference on Software Engineering*, pages 396–406, Singapore, Apr. 1988. IEEE Computer Society Press.
9. P. Khordoc, M. Dufresne, and E. Czerny. A Stimulus/Response System based on Hierarchical Timing Diagrams, Publication No.770. Technical report, Universite de Montreal, 1991.
10. F. Korf. *System-Level Synthesewerkzeuge: Von der Theorie zur Anwendung*. Dissertation, Fachbereich Informatik, Carl von Ossietzky Universität Oldenburg, Germany, 1997.
11. C. Lewerentz and T. Lindner, editors. *Formal Development of Reactive Systems: Case Study Production Cell*, volume 891 of *Lecture Notes in Computer Science*. Springer-Verlag, Jan. 1995.

12. K. Lüth. The ICOS synthesis environment. In Ravn and Rischel [17], pages 294–297.
13. K. Lüth, A. Metzner, T. Peikenkamp, and J. Risau. The EVENTS Approach to Rapid Prototyping for Embedded Control Systems. In *Zielarchitekturen eingebetteter Systeme (ZES '97), 14. ITG/GI Fachtagung Architektur von Rechnersystemen (ARCS'97)*, Rostock, Germany, September 1997.
14. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*, volume 1. Springer-Verlag, 1992.
15. E.-R. Olderog and H. Dierks. Decomposing real-time specifications. In H. Langmaack, W. de Roever, and A. Pnueli, editors, *Compositionality: The Significant Difference*, Lecture Notes in Computer Science. Springer-Verlag, to appear 1998.
16. A. P. Ravn. *Design of Embedded Real-Time Computing Systems.* Doctoral dissertation, Department of Computer Science, Danish Technical University, Lyngby, DK, Oct. 1995. Available as technical report ID-TR: 1995-170.
17. A. P. Ravn and H. Rischel, editors. *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'98)*, volume 1486 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
18. W. Thomas. Automata on infinite objects. In J. v. Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 4, pages 133–191. North-Holland, 1990.
19. P. Vanbekbergen, G. Gossens, and B. Lin. Modeling and synthesis of timed asynchronous circuits. In *Proceedings EURO-DAC with EURO-VHDL 94.* IEEE Comp. Soc. Press, 1994.
20. Zhou Chaochen, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.

Object-Oriented Modelling and Specification using SHE

M.C.W. Geilen, J.P.M. Voeten
Section of Information and Communication Systems
Faculty of Electrical Engineering
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Abstract

Industry is facing a crisis in the design of complex hardware/software systems. Due to the increasing complexity, the gap between the generation of a product idea and the realisation of a working system is expanding rapidly. To manage complexity and to shorten design cycles, industry is forced to look at system level languages towards specification and design. The (formal) system level modelling language called POOSL is very expressive and is able to model dynamic hard real-time behaviour as well as static (architecture and topology) structure in an object-oriented fashion. The language integrates a process part, based on the process algebra CCS, with a data part, based on the concepts of traditional object-oriented programming languages. Currently a number of automated software tools (model editing, simulator and compiler tools) are available in an environment which is called SHESim. These tools allow visual entry of structure and topology of the system, whereas the behaviour of individual processes is expressed in an imperative textual language.

1 Introduction

Industry struggles with mastering the development process of complex hardware/software systems. Examples of these systems are data/telecommunication systems, multimedia systems, medical systems, industrial control systems and consumer electronics. Hardware/software systems are typically implemented using several types of hardware components and complex application software running on real-time operating systems. These systems perform very complex behaviour and have to satisfy architectural constraints that impose physical distribution and hardware/software partitioning. Currently, industry is facing a crisis in the design of these complex systems [5]. Due to their combination of requirements, modern hardware/software systems are inherently complex to design, implement, verify, validate, test and debug. Today's systems integrate more and more functionality that has to satisfy stringent requirements of quality, flexibility, reliability and reusability. Deciding on an appropriate implementation has become increasingly difficult due to the diversity of available technology components (microprocessors, digital signal processors, application specific instruction processors, application specific integrated circuits, field programmable gate arrays, real-time operating systems, etcetera). Industry is struggling with tight bounds on time-to-market, -quality and -volume, high innovation speed and high costs for failure.

Existing specification and design approaches are no longer keeping pace with the growth of system complexity. For hardware, traditional design approaches are well established for the lower levels (transistor, logic, register and algorithmic level) of the design trajectory. However, as systems become more complex, the descriptions on these abstraction levels grow so huge that they become incomprehensible for designers and unmanageable for design tools [2, 5]. For the development of software, methods supporting the higher levels of the design trajectory are available. These methods, however, are often unsuitable for system design, and they do not help in making design decisions regarding architecture structure, hardware/software partitioning and the technology components to be applied.

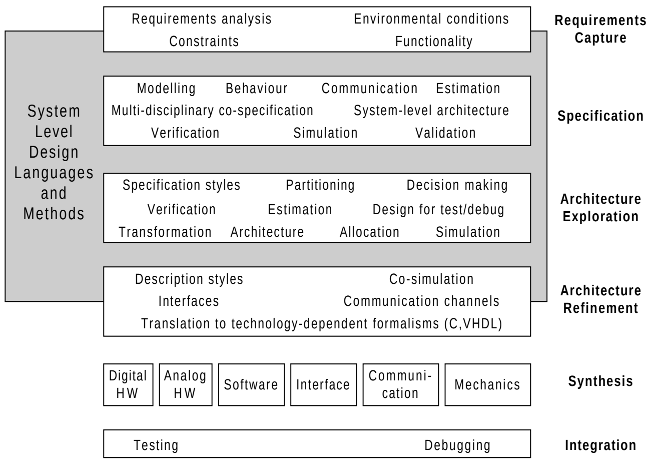


Figure 1. The role of system level languages

Due to the increasing system complexity, the gap between the generation of a product idea and the realisation of a working system is expanding rapidly. To manage complexity and to shorten design cycles, industry is forced to look at system level approaches towards specification and design [2, 5]. System level specification and design methods allow the creation of formal, executable models describing a system in the earliest stages of the design process. These models are expressed in powerful languages such that complex systems can be described with relatively few language constructs. System level models allow the properties of a system to be analysed, simulated, validated and verified before this system is actually being realised in terms of hardware and software components [2]. In this way design errors can be detected in a very early phase, thereby preventing expensive and time-consuming design iterations. The models further allow the systematic and rapid evaluation of different hardware/software architectures on a high level of abstraction, without having to spend a lot of time in creating low level implementations. This is an important advantage since it is at the higher levels of abstractions where the largest gains in designing 'optimal' hardware/software systems can be obtained [2].

In this paper we report on the system level modelling language called POOSL (Parallel Object-Oriented Specification Language). In Section 2 our requirements for a system level language are pointed out. Section 3 explains the basic concepts of POOSL and compares them to the concepts of the description languages SDL, Estelle, LOTOS and ROOM. In Section 4 we show how the language can be used to model a non-trivial part of a datalink protocol. Section 5 describes the software tools that are currently available. It will be shown how these tools allow the estimation of a performance parameter of this protocol. Finally in Section 6 we present our conclusions and directions for future research.

2 System Level Language Requirements

Figure 1 shows the role of system level languages and methods in the design trajectory for hardware/software systems [5]. It visualises the different design phases and indicates for each phase the relevant keywords and key activities. It should be clear that system level languages are especially useful in the early phases of design, where a system is being analysed and specified and where the system level architecture is being decided upon. They should help to bridge the gap between requirements capture and synthesis. Therefore they should be intuitive, applicable within multidisciplinary design and easy to understand. For this reason, a system level language should have an imperative nature, should be simple and built on a small collection of blending language primitives [1]. During the architecture exploration phase, the implementation technologies to be applied are decided upon. To make architecture exploration feasible, a system level specification should not be biased towards a particular choice of implementation and should be described in a language that abstracts from technology. To help designers bridge the gap between requirements capture and synthesis, adequate tool support is indispensable. Tools are required to edit, validate, verify, simulate and transform specifications, but are also needed for parameter estimation, automatic test-suite generation and system level synthesis to compile specifications to technology-specific languages such as VHDL and C. To support the development of such tools, a system level language should be equipped with a mathematical semantics [1].

From the characteristics of complex hardware/software systems, a number of other language requirements can be deduced. The communication and functional behaviour of a hardware/software system is often so complicated that it is quite impossible to understand it or to model it in its entirety. Therefore behaviour should be distributed over modules whose behaviour can be understood without having to understand the environment they are placed in. This requires modules to be self-contained, autonomous, relatively independent, and weakly coupled entities. We will call these entities process objects, or processes in short. Processes exist in their own right, have their own responsibilities, and they have their own activities to perform. This is highly desirable because it enables processes to be reused in contexts other than the one for which they were originally devised. Reuse is achieved by organising objects into classes. A class is a template from which objects are instantiated.

A very important characteristic of hardware/software systems is structure. Physical or spatial distribution, physical topology, hardware/software partitioning and software layering are all examples of phenomena that impose structure. The modelling of topology requires processes to be structured in a network of channels. The other forms of structure require the possibility to group processes into higher-order entities, called clusters. A cluster acts as an hierarchical abstraction of its internals. Hierarchy is an important concept that helps to manage complexity. To enhance modularity and reuse, clusters should be organised into classes.

Processes (and clusters) are relatively independent and they can perform their activities concurrently and at their own speed. Despite of their autonomy, processes will frequently have to exchange information. To make the coupling between processes (and clusters) as weak as possible, information exchange should be based upon message passing. The essential feature of message passing is that information is exchanged by means of an intermediate artefact, called a message. The purpose of a message is to reduce the coupling between the message senders and the message receivers [12]. To exchange information, the sender and the receiver only have to share the format and the general semantics of a message. They do not have to know anything about each other's internals, i.e. about private data. The internals of a process are said to be encapsulated by a strong encapsulation boundary. The only way for other processes to access these internals is through a clear-cut message interface.

A process can exhibit very complex behaviour. It can perform several activities in parallel and is able to exchange messages in various ways (synchronous, asynchronous, interrupt-driven and by broadcast). Behaviour can depend on (hard) real-time concepts such as time-outs, watchdogs and deadlines. It must be possible to express all these forms of behaviour in a system level modelling language.

Processes in hardware/software systems are able to perform complex manipulations on their private data. Because private data can have complex structures of their own, they should be represented by objects too. We will call these objects data objects. Data objects can be exchanged as parameters of messages between processes. Within a single process, data objects are shared and several activities can be manipulating these data objects concurrently. Different processes cannot share any data objects. To make data objects reusable, they have to be organised into classes.

3 Basic Concepts of POOSL

In [15, 1] we introduced the method Software/Hardware Engineering (SHE). SHE is an object-oriented modelling technique. Starting from informal graphical models, SHE produces rigorous behaviour and architecture models expressed in the formal specification language POOSL [1, 3]. The POOSL language has specifically been developed to be used as a system level modelling language.

The POOSL language combines a process part with a data part. The process part is based on the key ideas of the process algebra CCS [7]. The data part is based upon the concepts of traditional sequential object-oriented programming languages such as Smalltalk and C++. In POOSL very complex hard real-time behaviour is represented by a collection of asynchronous concurrent process objects that communicate synchronously or asynchronously by passing messages over static channels. Behaviour of process objects is described by

- synchronous (conditional) message (and data object) passing primitives;
- asynchronous (unbuffered) broadcast message passing primitives;

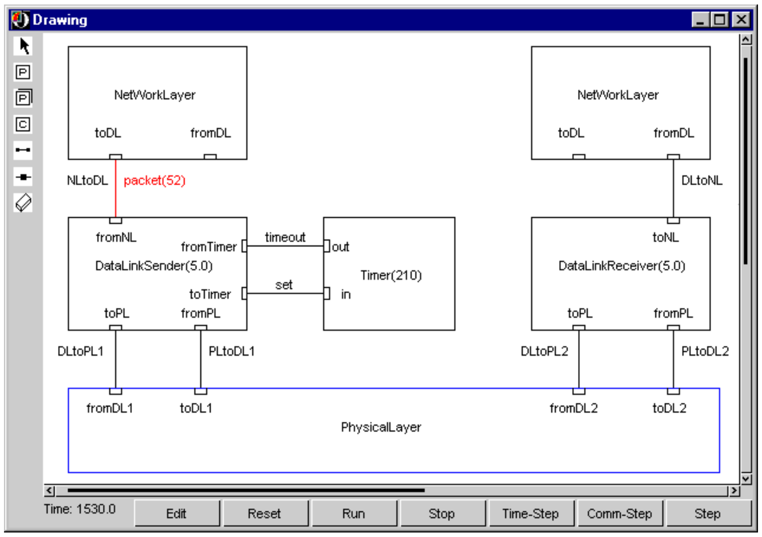


Figure 2. A Protocol Stack

- loop and select primitives, guarded commands, parallel and sequential composition, procedure (method) abstraction and (tail) recursion;
- interrupt and abort primitives;
- delay primitives.

Next to processes, POOSL supports the concept of cluster. A cluster is composed of processes and other clusters and acts as an abstraction of these. Clusters are used to create an hierarchical structure of modules that hide their internal structure. The constituents of a cluster are composed by parallel composition, channel hiding and channel renaming. These combinators are based upon similar combinators originally used in CCS [7]. Together, clusters and channels are suitable for describing architecture structure, topology and implementation boundaries [1].

To describe complex functional behaviour, POOSL supports data objects. Data objects have sequential behaviour and communicate by synchronous message passing. They are contained in processes and they model the private data of these processes. Data objects are also called travelling objects, since they can be passed between processes.

The POOSL language is equipped with a complete mathematical semantics. The semantics is based on a two phase execution model. The state of a system can either change by asynchronously executing atomic (communication or data processing) actions (taking no time) or by letting time pass (synchronously). The semantics of the non real-time part of POOSL is given in [1]. The formalisation of the real-time extension is described in [3, 4].

The key feature of POOSL is the expressive power to model very complex hard real-time (communication and functional) behaviour as well as static (architecture and topology) structure in an object-oriented fashion. POOSL can best be compared with the formal description languages LOTOS, SDL and Estelle [14] and the modelling language ROOM [12]. These languages all have a well-defined semantics[1] and are relatively simple[2]. They are further based on autonomous, encapsulated asynchronous concurrent entities that are connected in a static topology of channels and communicate using well-defined interaction primitives[3]. However, there are many concepts in which these languages differ from POOSL. Some important differences are:

[1]The semantics of LOTOS and SDL are defined in a formal mathematical way. As far as we have been able to verify, this is not true for Estelle and ROOM.

[2]This especially holds for Estelle. The other languages are a lot more complicated.

[3]LOTOS supports concurrent processes. Processes in LOTOS are pieces of behaviour, but they are not objects that perform this behaviour. Nevertheless, objects can be modelled by processes [9].

Communication: Communication in SDL, Estelle and ROOM, is buffered asynchronously[4]. Buffered asynchronous message passing is certainly a useful concept and should be supported, but it is not sufficient for hardware/software systems. We have experienced that especially at higher levels of abstraction, the concept of synchronous communication[5] is indispensable. It should be noted that asynchronous interaction can easily be expressed in terms of synchronous interaction. This is not true in the other direction. Synchronous interaction is incorporated in the LOTOS language. Although it is based on undirected (multi-way) action synchronisation, and not message passing, it can be used to express directed message passing too [9], be it in a less natural way. None of the languages above support asynchronous broadcast primitives.

Interrupts and aborts: Interrupts and aborts are very important for describing complex behaviour. In case of an interrupt the current course of behaviour is temporarily suspended until the interrupting behaviour has terminated. In case of an abort the current course of behaviour is terminated and the aborting behaviour is started. The LOTOS language only supports aborts. In ROOM behaviour can both be aborted as well as interrupted. SDL and Estelle do not have interrupt or abort facilities.

Data Objects: For describing intricate dynamic data structures, the concept of data object is indispensable. Except for ROOM, none of the languages support data objects. LOTOS and SDL use abstract data types and Estelle uses Pascal types to model data. Abstract data types are attractive from a mathematical point of view, but they have appeared to be difficult to understand [10] and they are hard to simulate and implement efficiently. Pascal is a procedural language and has insufficient support for encapsulation and data modularisation.

Concurrency and Sharing: The combination of concurrency and shared data is often considered dangerous since it introduces a lot of non-determinism in models and can lead to synchronisation and mutual exclusion problems. We agree with this, but we think a restricted combination of concurrency and sharing is necessary to model real-life systems. Therefore, POOSL has a par-statement that allows concurrency inside processes. The restriction lies in the fact that operations on data objects are atomic and hence concurrency in processes is defined at a rather coarse grain. In this way mutual exclusion and synchronisation problems can be solved very naturally, without forbidding activities to operate on shared data entirely. The grain of concurrency in ROOM, SDL, Estelle and LOTOS is at the level of process objects[6]. These processes cannot share data and therefore it is not possible to model activities that operate on a piece of shared data concurrently.

Real-time: None of the above languages have sufficient support to model (hard) real-time behaviour. The LOTOS language does not have any facilities to express time at all[7]. Estelle supports a restricted timing mechanism in the form of time delays. However, the Estelle semantics restricts the interpretation of time in the weakest way possible, placing only those requirements that derive from assuming that time moves forward consistently for all modules in the same subsystem [14]. SDL has a built-in real-time mechanism, relying on an asynchronous timer mechanism that is able to access a global clock referring to the current moment in time. However the delay between the moment of timer expiration and the moment at which the SDL model reacts to this expiry is unbounded. Therefore SDL is not suitable for expressing hard-real time behaviour [6]. In fact, the concept of time is not an inherent property of a model in SDL but of the possibly underlying execution engine or operating system. The same holds for the ROOM language.

4 Example

In [17] we have shown the suitability of the SHE method and POOSL language for the *specification* and *design* of an industrial distributed control system. In this section we will demonstrate how the POOSL language can be used to *model* a non-trivial part of a protocol stack [13]. The protocol stack is shown in the simulator window of Figure 2. The stack consists of two peer *NetworkLayer*s, a *DataLinkSender* with a *Timer*, a *DataLinkReceiver* and a *PhysicalLayer*. These entities communicate by exchanging messages over channels. The *NetworkLayer*s exchange *Packets*. The left *NetworkLayer* can send messages of the form *packet(somePacket)* to the corresponding *DataLinkSender* by executing *toDL!packet(somePacket)* statements[8]. *toDL* is a port of the *NetworkLayer* object

[4]A sender can always send a message without knowing the readiness of the receiver. In general, messages are buffered until they are consumed by the receiver, but under some circumstances they can also be discarded.

[5]A sender may only send a message if the receiver is ready and willing to receive it.

[6]In ROOM these objects are called actors, in SDL process instances and in Estelle they are called modules.

[7]Although many proposals have been written to extend LOTOS with a notion of time.

[8]The behaviour descriptions of the *NetworkLayer*s, *DataLinkSender*, *Timer* and *DataLinkReceiver* are not given in this paper.

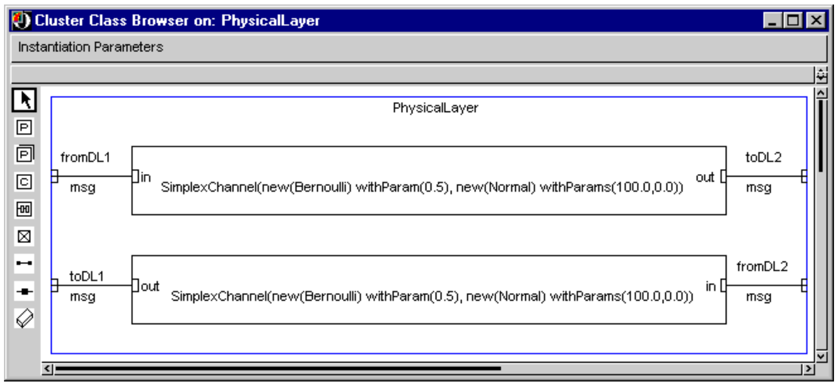

Cluster Class Browser on: PhysicalLayer
Instantiation Parameters
PhysicalLayer
fromDL1
msg
in
SimplexChannel(new(Bernoulli) withParam(0.5), new(Normal) withParams(100.0,0.0))
out
toDL2
msg
toDL1
msg
out
SimplexChannel(new(Bernoulli) withParam(0.5), new(Normal) withParams(100.0,0.0))
in
fromDL2
msg

Figure 3. PhysicalLayer cluster

and this port is connected to channel *NLtoDL*. The *DataLinkSender* can receive these *packet(somePacket)* messages by executing *fromNL?packet(somePacket)* statements. Here *somePacket* denotes a data object of data class *Packet*. Upon reception of a *Packet* the *DataLinkSender* wraps this *Packet* (together with some control information) in a data object of class *Frame*. This *Frame* is then sent to the *PhysicalLayer* which transports it to the *DataLinkReceiver*. Consequently, the *Packet* is retrieved from the *Frame* object, and is delivered to the peer *NetworkLayer*.

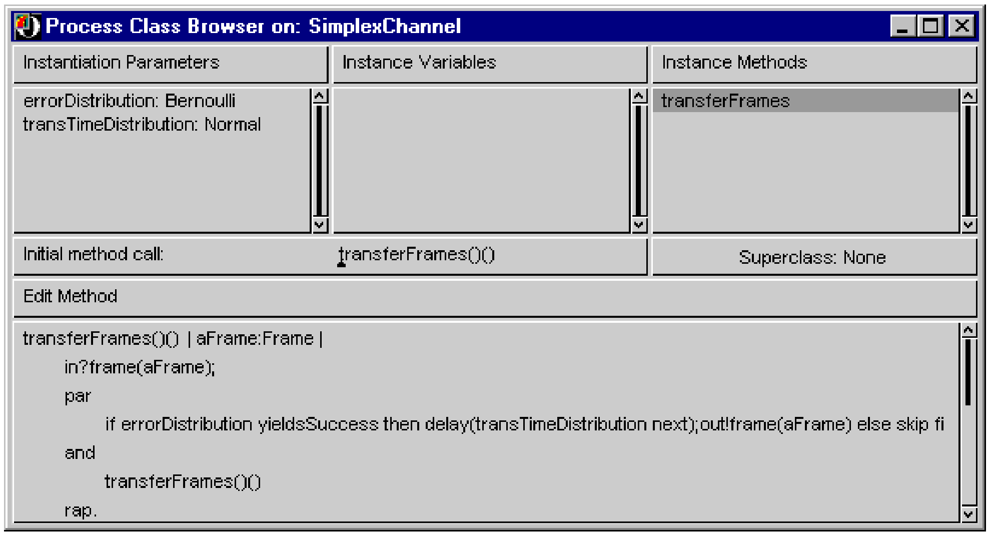
The *PhysicalLayer* is unreliable and can lose messages. To make sure that all *Packets* are delivered in the correct order, the *DatalinkSender* and *DataLinkReceiver* make use of a positive acknowledgement with retransmission protocol [13]. Therefore, they can send particular *Frames* several times and they can also send acknowledgement *Frames*. To this end the *DataLinkSender* makes use of a *Timer* process.

The specifications of the *DataLinkSender*, *Timer* and *DataLinkReceiver* are not further elaborated in this paper. Instead we will take a closer look at the specification of the *PhysicalLayer*.

PhysicalLayer is a cluster consisting of two *SimplexChannels*, see Figure 3. A *SimplexChannel* has two ports *in* and *out*. From port *in* it receives *frame(aFrame)* messages and after some specified period of time it delivers these messages at port *out*. The specification of the behaviour of a *SimplexChannel* is shown in the Process Class Browser of Figure 4. Within the *Instantiation Parameters* pane, two instantiation parameters, *errorDistribution* and *transTimeDistribution*, are specified. These parameters are bound at the moment a process of class *SimplexChannel* is instantiated. Parameter *errorDistribution* is a data object of data class *Bernoulli* and it models the probability that a message in the channel is lost. The parameter is bound to expression *new(Bernoulli) withParam(0.5)* which delivers a new Bernoulli distribution with success parameter 0.5 (half of the messages are lost), see Figure 3. The *transTimeDistribution* parameter models the transmission time of a message through the channel. This parameter is bound to a *Normal* distribution object with mean 100.0 and variance 0.0 (the transmission time is constant).

The behaviour of a process is specified in terms of *instance methods*. Methods can be compared to procedures of traditional imperative programming languages such as C or Pascal. Upon instantiation of a process, it calls its *initial method*. The initial method of class *SimplexChannel* is *transferFrames()()*. The definition of this method is given in the *Edit Method (POOSL)* pane of Figure 4. The definition starts with header *transferFrames()()*. The brackets indicate that methods can have input and output parameters. Clause | *aFrame:Frame* | declares a local method variable *aFrame* of data class *Frame*. The declaration of local variables is followed by the actual body of the method. Message-receive statement *in?frame(aFrame)* indicates that the process wants to receive (?) message *frame* with parameter *aFrame* from channel *in*. This statement is blocking; it is executed only if some other process executes a corresponding message-send (!) statement of the form *in!frame(someFrame)*.

After the reception of message *frame(aFrame)* a *SimplexChannel* executes two statements in parallel. Parallel composition is indicated by the *par* $\cdots$ *and* $\cdots$ *rap* construct. The first statement of the parallel composition starts by drawing a sample from the Bernoulli distribution and by checking whether this sample denotes success. This operation is performed by sending the message *yieldsSuccess* to data object *errorDistribution*. If this data object replies with *false* the message is lost (statement *skip* is executed). Otherwise, the *SimplexChannel* draws a sample



errorDistribution: Bernoulli
transTimeDistribution: Normal

transferFrames

transferFrames()()

```
transferFrames()() | aFrame:Frame |
    in?frame(aFrame);
    par
        if errorDistribution yieldsSuccess then delay(transTimeDistribution next);out!frame(aFrame) else skip fi
    and
        transferFrames()()
    rap.
```

Figure 4. SimplexChannel specification

from the Normal distribution *transTimeDistribution* by sending it the message *next*. Consequently, the channel starts to delay for the sample's amount of time units. After this delay, message-send statement *out!*frame(aFrame)* is executed. The !* symbol denotes that the message is sent asynchronously, i.e. the message is lost if no process is currently willing to receive it. The second statement *transferFrames()()* of the parallel composition is a method call. This statement starts the method, that is currently being executed again in parallel. In this way, the channel is able to transport an unbounded number of incoming messages.

The parallel activities in the *SimplexChannel* do not require any shared data. We could however introduce an instance variable *amountOfFrames* that stores the amount of *Frames*, currently being transported by the channel. Such a variable could be read or assigned a value by any concurrent activity at any instant in time. The modelling languages we discussed in Section 3 do not allow such behaviour to be expressed.

Notice that the behaviour of the *SimplexChannel* is far from trivial. Due to the expressive power of POOSL however, the behaviour can be expressed in only a few lines of easily readable code. This expressive power makes POOSL especially suitable for describing (the behaviour of) complex systems at a high level of abstraction without having to write detailed implementations in some hardware or software description languages such as VHDL or C (the reader is challenged to describe the behaviour of the channel in one of these languages).

5 Tool Support

The drawings we used in the previous section are snapshots taken from an interactive model editing and simulation tool for the SHE method. The tool is used to incrementally specify and modify classes of data, processes and clusters. A specification does not have to be complete before it can be tested and simulated. Using the different buttons on the bottom of the *Drawing* window of Figure 2, the current (partial) model can be executed in different modes of simulation. The messages and parameters that are passed between the different processes and clusters are indicated on the appropriate channels. It is possible to open inspectors on each model part (data objects, process objects and clusters). The inspectors show the current state of each variable; for processes they show the code that is currently being executed as well.

To inspect the history of all messages exchanged between the different entities, interaction diagrams (also called message sequence charts) can be generated automatically. An example of an interaction diagram is shown in Figure 5. The diagram shows the different model entities and the messages exchanged between these entities. For each message, the time, the channel and the parameters are indicated. For instance on time 621.0, message *packet* with parameter 2 (this is the sequence number of the corresponding *Packet* data object) is exchanged between the left-most *NetworkLayer* and *DataLinkSender*. Notice that not every instance of message exchange is shown in Figure 5.

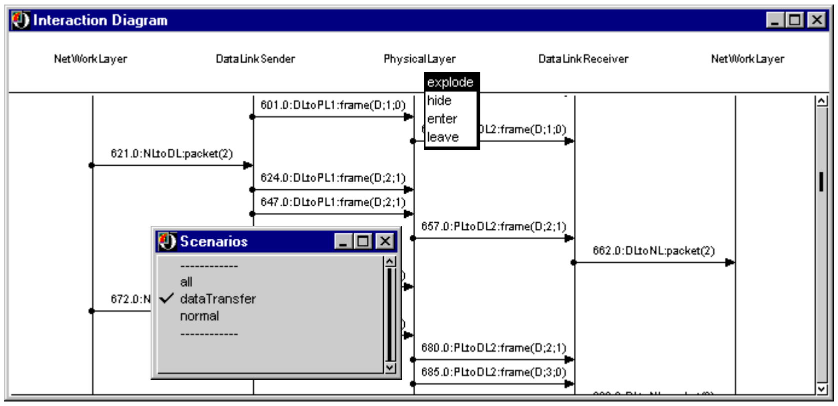


Figure 5. Interaction Diagram

The communications inside the *PhysicalLayer* are hidden. They can be made visible, however, by either exploding or entering the *PhysicalLayer*, see Figure 5. Another way to handle the complexity of behaviour is through the concept of *scenario* [1, 11]. A scenario defines a coherent piece of behaviour in terms of the entities (processes, clusters and channels) that are involved in this behaviour. The interaction diagram of Figure 5 visualises the behaviour corresponding to the *dataTransfer* scenario only. This scenario involves the two *NetworkLayer*s, the *DataLinkSender*, the *DataLinkReceiver* and the *PhysicalLayer* as well as channels *NLtoDL*, *DLtoPL1*, *PLtoDL2* and *DLtoNL*. The other entities are invisible.

Using the Interaction Diagrams we have been able to manually estimate the time-out time (of the *Timer*) that establishes an optimal (with respect to throughput) link between the *NetworkLayers*[9]. Counter-intuitively, this optimal time-out appeared to be lying around 27 time units and yields a throughput of 1 packet every 284 units of time. Of course, having to estimate performance figures manually is not desirable. Therefore an important topic of our research concerns the (statistical) estimation of model parameters (such as performance) automatically.

By using a formal verification tool for CCS, we have been able to verify the correctness of an abstract CCS version of the protocol. Even for the relatively simple positive acknowledgement with retransmission protocol we had to make severe abstractions (with respect to real-time and unbounded sequence numbers of packets) to make formal verification feasible. The truth is that the general problem of exhaustive formal verification is either theoretically or practically unsolvable. Therefore we are investigating methods to perform automatic, non-exhaustive verification on full models. In our opinion this is an important and challenging new research topic.

Research is also being carried out into system level synthesis of POOSL specifications. The first results into the mapping of POOSL onto hardware are presented in [8]. For the automatic mapping onto software (C++), promising results have been obtained. Further a system of behaviour-preserving transformations has been developed [1, 16]. Behaviour-preserving transformations allow the architecture of POOSL models to be modified, without changing the functional and communication behaviour.

6 Conclusions and Future Research

In this paper we have described the system level modelling language POOSL. POOSL is a language with a complete mathematical semantics and is developed as part of the SHE (Software/Hardware Engineering) method for hardware/software systems. The language combines a process part based on CCS, with a data part based on the concepts of traditional object-oriented programming languages. Very complex hard real-time behaviour is represented by a collection of asynchronous concurrent process objects that communicate synchronously by passing messages (with data objects) over static channels. Behaviour of process objects is described by synchronous message passing primitives, choice, loop and select primitives, guarded commands, parallel and sequential composition, method abstraction, (tail) recursion and interrupt, abort and delay primitives. To describe intricate data structures,

[9]In this experiment we have used a slightly more complex *PhysicalLayer*.

POOSL supports data objects. Data objects are contained in processes and they model the private data of these processes. A restricted form of concurrency on shared data within a process is allowed. Architecture structure and topology of systems can be represented by clusters and channels. A cluster is composed of processes and other clusters and acts as an abstraction of these. The combination of supported concepts makes POOSL a powerful system level language that has important avantages over other modelling languages such as SDL, Estelle, LOTOS and ROOM.

A number of supporting software tools have been developed. An available interactive simulator tool allows POOSL models to be entered, simulated and validated. Model validation is supported by interaction diagrams and scenarios. Interaction diagrams also support the manual performance evaluation models. By making abstractions into process algebra CCS, a limited form of formal verification can be carried out. Important current research topics involve the automatic non-exhaustive verification of (hard real-time) properties, statistical estimation of performance parameters and system level hardware and software synthesis.

References

[1] P. V. der Putten and J. Voeten. *Specification of Reactive Hardware/Software Systems.* PhD thesis, Department of Electrical Engineering, Eindhoven University of Technology, Eindhoven, The Netherlands, 1997.

[2] D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems.* Prentice-Hall, Englewood Cliffs, 1994.

[3] M. Geilen. Real-Time Concepts for Software/Hardware Engineering. Master's thesis, Faculty of Electrical Engineering, Eindhoven University of Technology, Eindhoven, The Netherlands, 1996.

[4] M. Geilen and J. Voeten. Real-Time Concepts for a Formal Specification Language for Software/Hardware Systems. In J. Veen, editor, *Proceedings of ProRISC/IEEE'97. Utrecht : STW, Technology Foundation*, 1997.

[5] J.P.M. Voeten, P.H.A. van der Putten, M.C.W. Geilen, H.P.E. Vranken and M.P.J. Stevens. *System Level Description of Complex Hybrid Systems. Presentation at Barga System Level Design Workshop, Barga, Italy*, July 8–10 1997. Proceedings of workshop are available on the web: http://www.ecsi.org/ecsi/ecsi.html.

[6] S. Leue. Specifying Real-Time Requirements for SDL Specifications - A Temporal Logic-Based Approach. In P. Dembiński and M. Średniawa, editors, *Protocol Specification, Testing and Verification XV*, pages 19–34, London, 1997. Chapman & Hall.

[7] R. Milner. *Communication and Concurrency.* Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[8] R. Michielsen and J. Voeten, Implementation of POOSL in Hardware, in *Proceedings of ProRISC/IEEE'97. Utrecht : STW, Technology Foundation*, J.P. Veen, Ed., 1997.

[9] A. Moreira. *Rigorous Object-Oriented Analysis.* PhD thesis, University of Stirling, Scotland, 1994.

[10] K. Narfelt. SYSDAX : An Object Oriented Design Methodology Based on SDL. In R. Saracco and P. Tilanus, editors, *Proceedings of SDL'87: State of the Art and Future Trends*, pages 247–254, Amsterdam, 1987. North-Holland.

[11] P. van der Putten, J. Voeten, M. Geilen and M. Stevens. Multidisciplinary scenarios in hardware/software engineering, object-oriented co-specification of complex systems. In J. Veen, editor, *Proceedings of ProRISC/IEEE'97. Utrecht : STW, Technology Foundation*, 1997.

[12] B. Selic, G. Gullekson, and P. Ward. *Real-Time Object-Oriented Modeling.* Wiley & Sons, New York, 1994.

[13] A. Tanenbaum. *Computer Networks 3rd ed.* Prentice-Hall, Englewood Cliffs, 1996.

[14] K. Turner. *Using Formal Description Techniques.* Wiley & Sons, Chichester, 1993.

[15] P. van der Putten, J. Voeten, and M. Stevens. Object-Oriented Co-Design for Hardware/Software Systems. In M. Cavanaugh, editor, *Proceedings of EUROMICRO'95*, pages 718–726, Los Alamitos, California, 1995. IEEE Computer Society Press.

[16] J. Voeten, P. van der Putten, and M. Stevens, Behaviour-Preserving Transformations in SHE : A Formal Approach to Architecture Design, in *Proceedings of EUROMICRO'96*, P. Milligan and K. Kuchcinski, Eds., Los Alamitos, California, 1996, pp. 19–27, IEEE Computer Society Press.

[17] J. Voeten, P. van der Putten, and M. Stevens. Systematic Development of Industrial Control Systems using Software/Hardware Engineering. In P. Milligan and P. Corr, editors, *Proceedings of EUROMICRO'97, Short Contributions*, pages 26–36, Los Alamitos, California, 1998. IEEE Computer Society Press.

uBET: Graphical Specification Support in an Industrial Environment

Bart Knaack
Lucent Technologies, The Netherlands

Abstract

The usage of formal methods and design tools in the software development process is becoming a key issue in the industrial software development world. In this light we studied the Lucent Technologies software design process. We describe the MSC-requirements description method and investigate how the usage of MSC can aid in the software design process. Furthermore we look at the uBET-tool for the support of MSC deployment and compare the uBET syntax and semantics with the formal MSC definition.

L. Feijs
Philips Research Laboratories,
Eindhoven University of Technology
R. Krikhaar
Philips Medical Systems
R. van Ommering
Philips Research Laboratories

Formalization and Visualization of Software Architectures

Abstract

Software architectures play an important role in the development of software intensive systems. Examples of such systems are medical systems (X-Ray, MRI scanners), telecommunication systems (telephony, videocommunication) and consumer systems (television, VCR).

Different views have been recognised to split up the complex task of defining software architectures: execution view, logical view, code view, physical view and module view. In our presentation we focus on how parts of the module view of software architecture can be formalised and visualised. For this purpose we have developed Relation Partition Algebra (RPA) which is an extension of relation algebra. In the last decade we had the opportunity to validate our ideas on RPA and its application at different development sites withing Philips.

Besides defining good software architectures one should also ensure that different results of software development are conform the defined software architecture. We have also applied RPA to introduce software *architecture verification* in (existing) development environments.

This abstract describes the lecture to be delivered by R. Krikhaar, whereas Sections 1 to 7 of the present text (participants proceedings) is essentially the paper "Architecture Visualisation and Analysis, Motivation and Example" by L. Feijs and R. Van Ommering, in slightly modified form presented at the ARES International Workshop on Development and Evolution of Software Architectures for Product Families 1996 (the lecture also covers more recent results and a new paper is being written).

1 Introduction and Motivation

Whereas for programming in the small there exist well-established concepts of specification and implementation, the subject of ‘programming in the large’ (or software architecture, as we sometimes call it), has hardly any established terminology. In this report we elaborate the idea that an architecture A is a specification of the intended structure of a large design, and a concrete design D is a structure ‘as realised’. We present some evidence that it will become possible to have a vocabulary to express A and the technology to verify whether a given D does satisfy A.

The software complexity of many Philips products increases, amongst other reasons because many features are to be realised in software instead of in hardware. Often there is a need to deal with product families instead of just products. Not only the processor performance grows exponentially, but most often the software size grows exponentially too. Other complicating factors are that closed boxes become part of open systems, that media become part of multi-media, and that in many business groups there is a growing interest in optimising software re-use.

In today’s software engineering practice, products go through a concept phase in which the architecture is well-visualised by means of diagrams, while the team is still growing, and while a good architect is present. But in the realisation phase the architecture diagrams may have become obsolete, and although low-level coding standards and analysis tools exist, it is usually not possible to check the real software against the high-level architecture. In the realisation phase, there can be serious problems in managing the software complexity. Yet there is hope: some visualisation techniques exist already and for example the tool TEDDY turned out to be already very useful for analysing evolving architectures; several other developments point into the same direction, for example QAC and Graphical Designer.

Some of the ideas reported in the present report were developed in the context of the project ‘design engineering methods’. We list some key ideas of this report:

- many relevant structures in a software architecture are nothing but binary relations,
- manipulating relations is a way of obtaining views on intended or concrete architectures,
- alternative views on concrete software architectures can be visualised,
- a modest amount of automated support could be of great help.

It is a goal of this report to explain the idea of software architecture verification. We believe that there are technical options which will help in understanding the evolving Philips architectures. It would be beneficial to have a kind of continuity in the architecture evolution in the sense that in all project phases there are explicit architecture rules, and up-to-date high level views which are kept consistent with the real software.

The main body of this report consists of a story about a fictitious software team which develops a fictitious product and applies some verification techniques to its evolving architecture. In order to keep this report as short as possible, the fictitious product is just a toy example, but nevertheless the example will convey the idea of software architecture verification. Sections 2 to 6 present this story. Please note that the data structures used to define the architecture as well as the structure extracted from the code are just examples.

Related work: The idea of architecture verification is also presented in [1] and [2]. The TEDDY tool is described until now only in Philips internal reports. There are also Philips internal reports containing a detailed study of the theory of relations for purposes of softwarearchitecting. For more information about the mathematical theory of relations we refer to [3].

Acknowledgements: The author would like to thank René Krikhaar for the cooperation on the subject of this report. In fact, this report is only an extremely simplified case study of things which the second author and René are already putting into practice in real-life systems. The author would like to thank Peter van den Hamer for the discussions and for his feedback on an earlier version of this report. Special thanks go to Henk Obbink, project

leader of the ‘design engineering methods’ project, which turned out to be a fruitful place for discussions and for the exchange of ideas.

2 The initial software architecture

In the beginning of the system's conception there is a software architect whose task it is to define the system's software architecture. The software architect will listen to all the marketeers, customer's representatives and hardware specialists. Of course there is a project document listing goals like flexibility, modularity, future-proofness as well as a list of performance requirements and cost constraints.

Performance requirements and cost constraints strongly influence the number and types of microprocessors, the programming language (assembly, C-like, C, C++, SDL, and so on) and the kind of operating system or RTK that is affordable. But the modularity goals are covered by a software architecture diagram. For a real-life 512 Kbyte system, say, this diagram may in the initial phase take the form of an A4-sized drawing with some 40 to 70 named boxes, organised in layers.

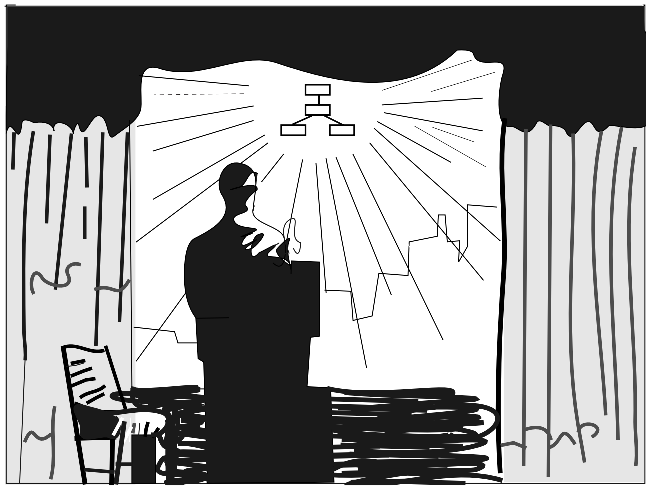


Fig. 1. The software architecture is presented.

In this paper we shall only follow a mock-up development process of a toy architecture, so instead of 40 boxes (software components) we look at a diagram of just 4. But we shall be explicit about the 'uses' relations between these software components.

So let us assume that after some time the software architect presents the key software components and a diagram of the designed component-level 'use' relation. He presents this to the designers and programmer(s) who are going to further detail this design and who will eventually make the C programs. Let us assume that amongst other things, the architect says:

> "Dear friends, Figure 2 is our software architecture: there are four software components, which I will explain now. `RSRC_MNGR` is the Resource Manager, which will contain the main procedures of all our processes and these will be scheduled by the HW and SW of our platform. `SYS_FUNC` contains the System Functions, and this is the heart of our system. This will provide the data transformations our customers are waiting for. `HW_ABSTR` is the minimal Abstraction of the special Hardware of our platform. `ERR_DRVR` is the Error Driver which provides for error printing and contains a driver for the special error LED. The lines in Figure 2, directed from top to bottom, show the 'use' relations foreseen. So for example from `SYS_FUNC` you are allowed to call the functions of `HW_ABSTR`."

Although here Figure 2 has been made by hand, it could also have been made with TEDDY, a browser which proposes an initial diagram layout after reading the file with the essential information of the ‘uses’ relation, and which allows the user (the architect) to modify the layout interactively.

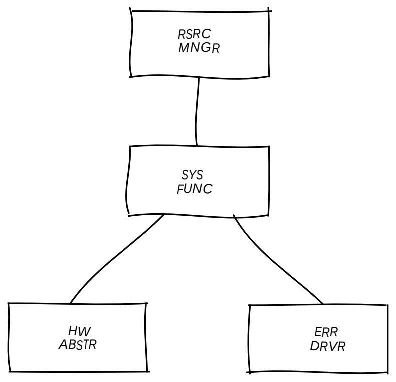


Fig. 2. Architected component-level ‘use’ relation.

Of course, instead of manual drawing or using TEDDY, other visualisation software could be used. The only requirement is that a diagram and an explicit binary relation are easily kept consistent. The best way of keeping two things consistent with each other is to generate one of them automatically from the other. It is important to note that the essential information of Figure 2 is an intended ‘use’ relations on components, which is as follows:

```
RSRC_MNGR SYS_FUNC
SYS_FUNC  HW_ABSTR
SYS_FUNC  ERR_DRVR
```

In many real projects, the processor is more powerful than the processor of the previous generation of products and there is two or four times as much ROM available, so most often the team is optimistic that that the software architecture is feasible. Also in our case, the programmers agree with the software architecture and happily they start filling in the details.

3 Details of the real system

After some work, the programmers will come up with C code and for example the error driver `ERR_DRV` could consist of two functions, `err_pr()` which calls `led_33()` and `led_33()` which calls `err_pr()`.

```
err_pr() { led_33(); }
led_33() { err_pr(); }
```

Since in this paper we are only following a mock-up development of a toy architecture, we shall not discuss the complex algorithmics of real-life software, nor the specification techniques necessary for that, but we only show some extremely simple C functions, calling each other, but with no meaningful functionality whatsoever. Note also that we do not stick to the usual layout conventions, in order to save space. After all these disclaimers, we give the C code of all software components.

```
/****************************************************
 *    Component: ERR_DRVR                           *
 ****************************************************/

err_pr() { led_33(); }
led_33() { err_pr(); }

/****************************************************
 *    Component: HW_ABSTR                           *
 ****************************************************/

#include "ERR_DRVR.h"

power() { err_pr(); i2c(); }
i2c()   {                   }

/****************************************************
 *    Component: RSRC_MNGR                          *
 ****************************************************/

#include "SYS_FUNC.h"
#include "HW_ABSTR.h"
#include "ERR_DRVR.h"

init()   { e(); led_33();            }
reboot() { power(); init(); power(); }
step()   { while (1+1==2) a();       }

/****************************************************
 *    Component: SYS_FUNC                           *
 ****************************************************/

#include "HW_ABSTR.h"
#include "ERR_DRVR.h"

a() { b(); c(); }
b() { power();  }
c() { d(); g(); }
d() { i2c();    }
e() { f();      }
f() { g(); err_pr();  led_33(); }
g() { h();                      }
h() { err_pr();                 }
```

4 Extracting relations from the real system

For a 512 Kbyte system, it may take a year from conception to completed code, and then the code is not as easily surveyed as the one page of C code of our toy example. After this year, there is no more focus on architecture, because everybody is busy with testing, debugging and adding shortcuts and tricks for meeting the performance requirements. New people joined the project and maybe the architect has already left.

If this were a real-life project and it would continue for yet another year, the project could find itself in a reverse engineering phase. There is even a danger that the project finds itself in the middle of a spaghetti: nobody understands all of the code and nobody understands the system's modular structure and the associated 'uses' relations.

But stop, we shouldn't wait until the spaghetti-phase. As soon as the initial code is available, it can be checked against the architecture diagram. Of course in a real-life project, there could be several levels of hierarchy, and there may be even more kinds of 'uses' relations than just the 'calls' relation on C functions, but the essential idea remains the same. It is possible to extract the real 'uses' relation, as opposed to the architected 'uses' relation from the code. All the information is on-line available; the only problem is that there is too much information. The obvious solution is to use automated support for extracting the essential information from the code. Again this information can be cast into the form of tables. There are technical options to perform this extraction, for example QAC, although of course they depend on the programming language in use.

For the present example the the essential 'use' information is easily extracted and stored in a file called `uses`.

```
err_pr  led_33
led_33  err_pr
power   err_pr
power   i2c
init    e
init    led_33
reboot  power
reboot  init
reboot  power
step    a
a       b
a       c
b       power
c       d
c       g
d       i2c
e       f
f       g
f       err_pr
f       led_33
g       h
h       err_pr
```

But it is not obvious that this satisfies the initial architecture of Figure 2. As a first step, it is already helpful to visualise this 'uses' relation, see Figure 3. This is made with a structure browser (TEDDY). Of course this browser is not only useful for analysing a design once finished, it could in principle also be used for drawing diagrams of relations which do not exist yet, but which will be implemented next. Note the thick line between `err_pr` and `led_33` which reveals the mutual usage (a thick line is a 'uses' arrow going upward, and since there is a cycle, there *must* be at least one thick arrow).

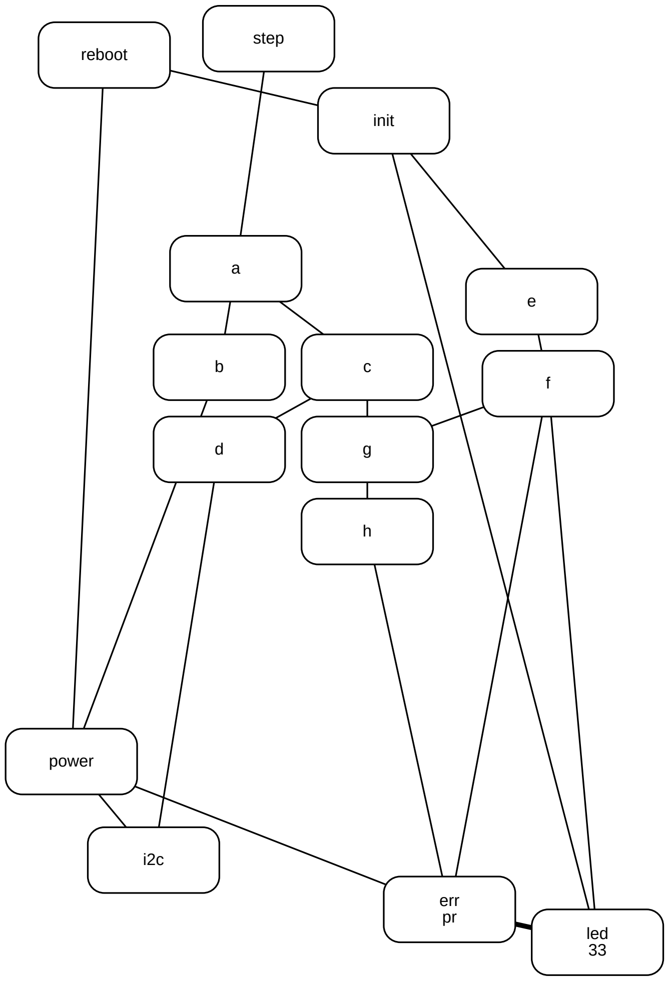


Fig. 3. The use relation at the C function level.

In order to apply a suitable process of abstraction of the ‘use’ relation we need the ‘part-of’ relation too. In this case, the ‘part of’ relation (which tells for each C function its component) is as given by the TEDDY diagram of Figure 4. This diagram shows the functions as boxes with rounded corners and the components (files) as rectangular boxes. Again, the essential information behind the ‘part of’ relation is just a binary relation; it can be stored as a file containing pairs of identifiers.

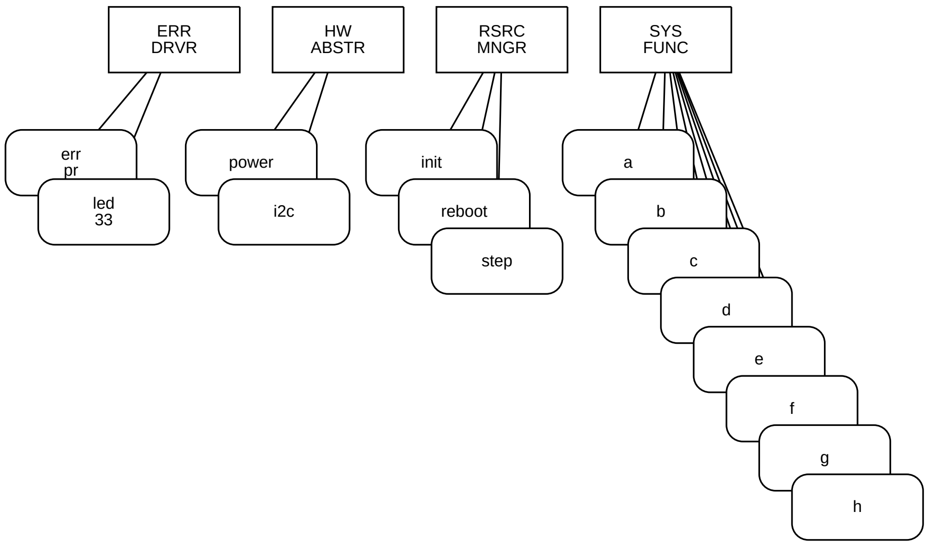


Fig. 4. ‘Part of’ relation between functions and components.

In particular, `err_pr` is part of `ERR_DRVR`. `led_33` is part of `ERR_DRVR`. `power` and `i2c` are part of `HW_ABSTR`. The function `init` is part of `RSRC_MNGR` and so are `reboot` and `step`. Finally, `a` to `h` are part of `SYS_FUNC`.

Now we have everything needed in order to compare Figure 2 and Figure 3. This is done by transforming the use relation from the C function level to a relation amongst software components. We call this transformation *lifting*: we move the relation from the level of the small objects (the C functions) to the level of the big objects (the software components). The key to this lifting is of course the ‘part-of’ relation of Figure 4.

In detail, the process of lifting goes as follows: from file `uses`, note that `err_pr` uses `led_33`. From the ‘part of’ relation (Figure 4), `err_pr` is in `ERR_DRVR` and `led_33` is in `ERR_DRVR` so `ERR_DRVR` uses itself, which is not so interesting. Conversely `led_33` uses `err_pr`, which adds no new information. Next, `power` uses `err_pr` and since `power` is in `HW_ABSTR` and `err_pr` is in `ERR_DRVR` we may conclude that `HW_ABSTR` uses `ERR_DRVR`. In the same way we find that `RSRC_MNGR` uses `SYS_FUNC`. When carrying along, the following relation is obtained; Assume that it is stored in a file called `Uses`.

```
HW_ABSTR  ERR_DRVR
RSRC_MNGR SYS_FUNC
RSRC_MNGR HW_ABSTR
RSRC_MNGR ERR_DRVR
SYS_FUNC  HW_ABSTR
SYS_FUNC  ERR_DRVR
```

This transformation of lifting, that is transforming a ‘uses’ relation to get a relation at a higher level, is a key concept for software architecture verification. It combines abstraction and advanced cross-referencing. The outcome is visualised in Figure 5.

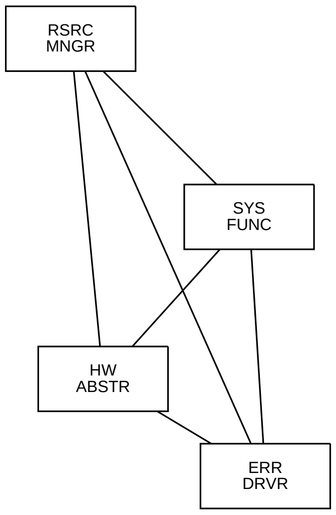


Fig. 5. The C-level use relation viewed at the component level.

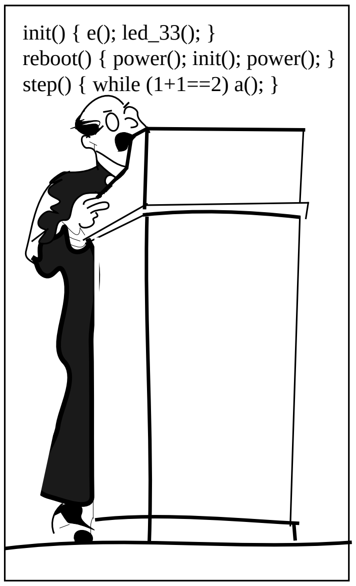
There is also another route to arrive at the same information, namely by looking at the `#include` lines in the C code. When using a UNIX system, it suffices to type `grep include *.c` and after some trvial post-processing we have a component-level ‘uses’ relation. Fortunately it is the same relation; if the `grep`ed relation would have had additional lines, this would indicate that there are superfluous `#include` lines (which happens not to be the case). The converse cannot happen at all, because if each component is compiled separately, the compiler will check that all C functions used are listed in one of the included header (`.h`) files; here we assume that the header files themselves are correct, in the sense that they list all headers of functions of their components, and nothing else.

Fig. 6. The architect discovers the real system.

Now it is time for a comparison. The team calls the architect and visualises the contents of the `Uses` relation, as in Figure 5. But look, Figures 2 and 5 are not the same. What has happened?

5 Discussion

Of course everybody wants to know what went wrong and why Figures 2 and 5 are not the same. Maybe the architect will say that the programmers have turned his clean architecture into a mess and maybe the programmers will say that the architect has not enough knowledge about *real* software.



```
init() { e(); led_33(); }
reboot() { power(); init(); power(); }
step() { while (1+1==2) a(); }
```

Fig. 7. The programmer explains why `RSRC_MNGR` must use `ERR_DRVR`.

But then the team realises that maybe there are no stupid mistakes at all and that maybe the problem is more subtle. One of the programmers explains an important observation first:

> The resource manager component `RSRC_MNGR` has three C functions, `init`, `reboot`, `step`, each of which can be viewed as an independent main program. Of these, `init` and `reboot` are tied to the hardware reset interrupt and the software interrupt (trap), whereas `step` is supposed to be called in an eternal loop. The architected component-level 'use' relation of Figure 2 has been made with the `step` function in mind. But everybody knows that for initialisation and rebooting one has to do some low level tricks every now and then. For example `reboot` has to call `power` and indeed, this causes a direct 'uses' line from `RSRC_MNGR` to `HW_ABSTR`. This explains why Figure 5 has more lines than Figure 2. And if you look at it this way, we have in fact respected the original architecture.

This seems a plausible explanation, but how can one be sure if this is really true? This demands a further analysis. There is a technique for analysing the code, namely by means of lifting. If one could (for the sake of the analysis) remove all 'uses' lines and all C functions not relevant to `step`, then one gets a thinned version of Figure 3. And then lifting could yield a thinned version of Figure 5.

It is clear that for small examples one can perform the lifting transformation manually, but for large systems automated support could be of great help. Let us assume that we have a piece of software which can do lifting of 'uses' relations and a few more related transformations. Let us call this piece of software a *relational calculator*. One of the main purposes of the present report is to explain the idea of a relational calculator as a technical option; we will not discuss any make-or-buy decisions, but we just assume we that we have it (also without the calculator many of its calculations are easily done by shellscripts or other ad-hoc programs).

A possible user-interface of the calculator is shown in Figure 8. The calculator works with binary relations, and just like a normal pocket-calculator it is important to show the outcomes of the calculations on some kind of display. Maybe one wants to choose between two kinds of display: a text-file format and a graphical format. The text-file display is trivial and the graphical display is already existing: use TEDDY or similar visualisation software.

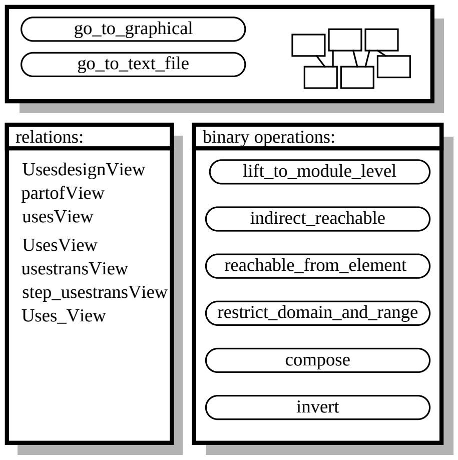


Fig. 8. The relational calculator.

The lower part of the user-interface consists of two halves. The left-hand side part allows for entering file-names for the storage and retrieval of relations. For example `Usesdesignview` is the name of Figure 2, `partofview` is the name of Figure 4, `usesview` is Figure 3, `Usesview` is Figure 5, and the remaining names belong to relation files which will arise soon if we continue the development of the toy architecture.

The right-hand side part allows has a number of buttons, one for each calculation which can be performed. For example in order to lift `Usesdesignview` of Figure 2 with the `partofView` of Figure 4, these two files must be selected and then the ‘lift_to_module_level’ button must be pressed.

6 Calculating a revised ‘Uses’ relation

Next the real analysis is performed, which begins with the removal of all ‘uses’ lines and all C functions not relevant to `step`. The first question is: ‘which functions are used by `step`?’ Somewhat more precise one wants to have all functions used by `step` as well as all functions used by such functions and so on. Therefore, one proceeds as follows: first calculate the transitive closure of the uses relation. This is exactly what the second button, labeled ‘indirectreachable’ is meant for. The outcome is visualised in Figure 9.

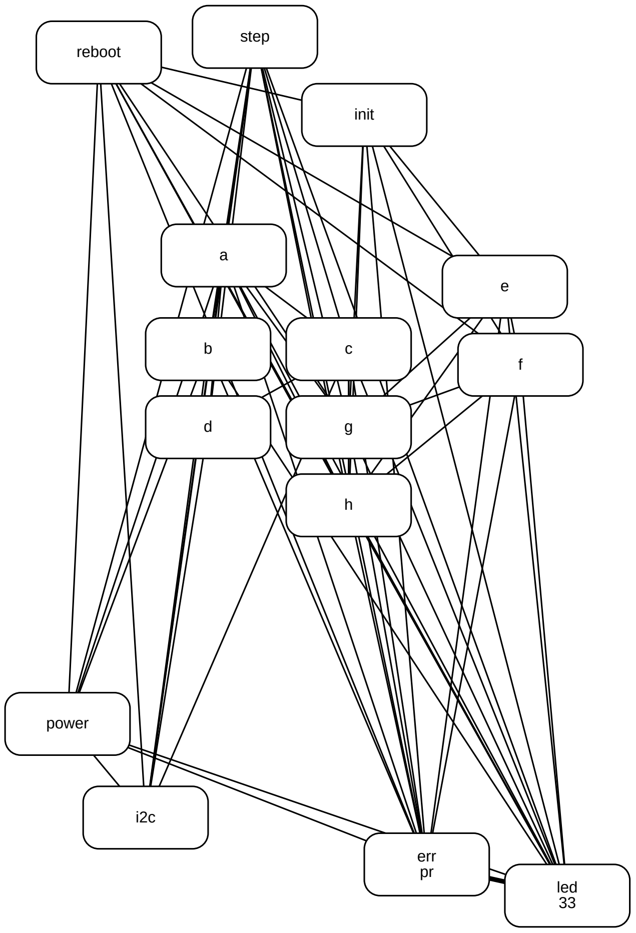


Fig. 9. Transitive closure of the ‘use’ relation on functions.

Next, this relation is restricted to those ‘uses’ pairs which begin with `step`, by means of the button ‘reachable_from_element’. The outcome is visualised in Figure 10.

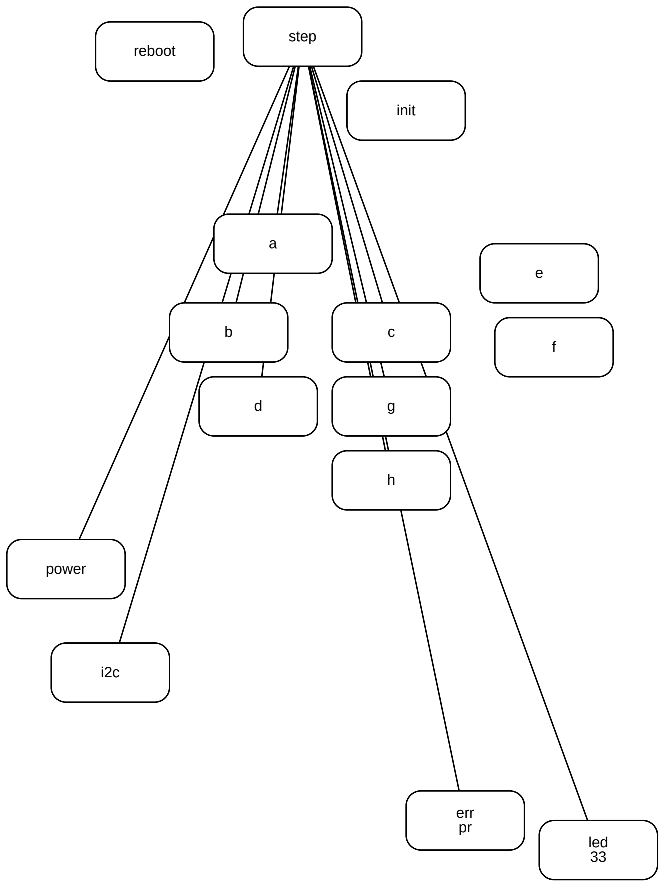


Fig. 10. Functions transitively connected to `step` function.

In fact this relation is not really interesting. The interesting things is the set of functions occurring in it. Let us assume that the calculator can work with sets instead of relations too (the buttons for that are not shown in Figure 8). Then the main result is the set:

```
a
b
c
d
err_pr
g
h
i2c
led_33
power
step
```

which means that `e`, `f`, `reboot` and `init` have been thrown out. This set must be used to restrict the ‘uses’ relation of Figure 3. Both the domain and the range of the latter ‘uses’ relation must be restricted. The button ‘restrict_domain_and_range’ makes the calculator perform this task. After that ‘lift_to_module_level’ must be applied to that result, which gives the main outcome of the analysis (Figure 11).

In this particular example, the same outcome would have been obtained by just removing `init` and `reboot` without removal of the functions which are not transitively connected to `step`, but in general, removal of such functions makes a difference, and therefore the latter part of the analysis may not be skipped. So now there is a main outcome, visualised in Figure 11.

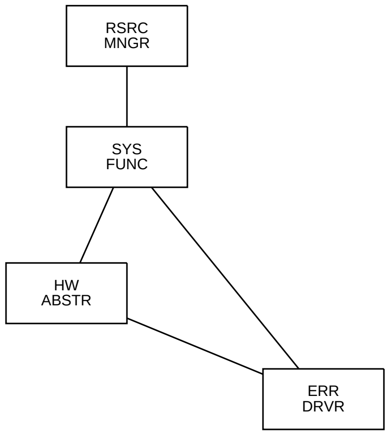


Fig. 11. Component level use relation with surpressed init and reboot calls.

Now it is clear that the programmer’s explanation is partially true, but not all of it. Indeed, most of the differences between Figures 2 and 5 are caused by `init` and `reboot`. But the ‘uses’ line from `HW_ABSTR` to `ERR_DRVR` is not explained that way. So the team must discuss this further. The team must arriving at one of two possible conclusions: either `HW_ABSTR` may not use `ERR_DRVR` and thus `power` should not invoke `err_pr` and should be modified; or it is really necessary that `power` invokes `err_pr` and thus the revised architecture of Figure 11 must be declared to be the official architecture. In this story, the latter alternative is chosen (see [4] for a nice dicussion of typical causes of such differences). By now, the team arrives at a common understanding of the architecture. The main lesson is that using some simple concepts about relations and a modest amount of automated support, several views on the system can be developed and compared.

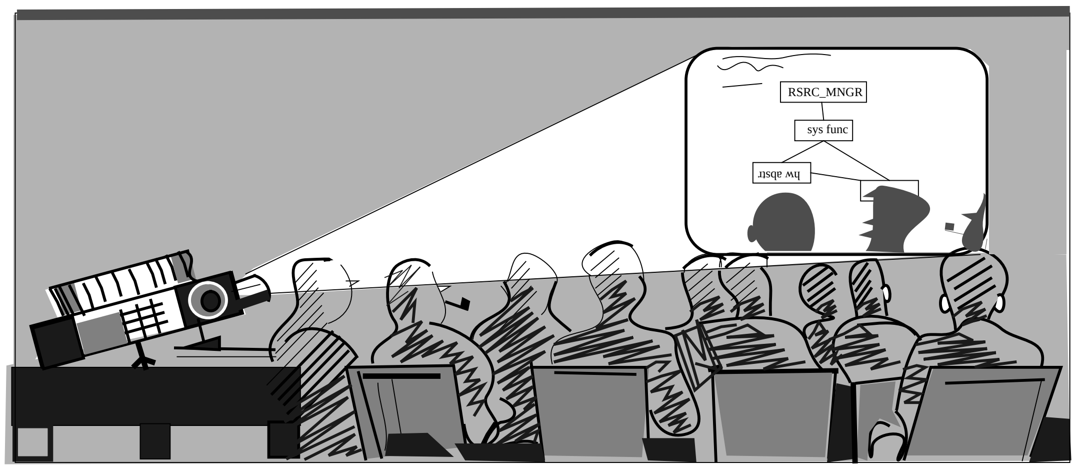


Fig. 12. The team arrives a common understanding of the architecture.

7 Concluding remarks

Of course the story and the case study of the preceding sections is extremely simplified and naive as compared to the real software development projects happening in Philips. But even for this toy architecture, a clear view on the transitive dependency amongst the C functions is not immediate: the graph of Figure 9 is already too complicated for being found manually or for being kept in mind during work. So can you imagine to find manually the transitive dependency amongst the C functions for a real product with 1000 C functions, or 10.000?

It was also unrealistic to assume that there are two kinds of interesting design objects (C functions and C files); and so was the assumption that there is only one kind of 'uses' relation ('uses', based on calling) and that there is only one dimension of decomposition ('partof' based on files). So in reality, when so much more relations exist, can you imagine trying to reconstruct the high-level architecture from the sources without any automated support?

In our opinion, the automated support should consist of three kinds of software:

- software for extracting relations from specifications and from source code,
- software for visualising structures and (relations),
- software for manipulating and combining relations in order to develop alternative views.

Software of the first kind will necessarily be specific for the languages in use. In certain cases some simple unix tools like awk and grep will do the job already. An example of software of the second kind is TEDDY. An example of software of the third kind is the calculator of Figure 8, which at present however has not been fully realised yet (Figure 8 was derived from a mock-up demo). There are still many open issues with respect to the ergonomical aspects of the calculator.

It can be argued that all this is only just one aspect of architecture, and that the other aspects of the system's behaviour are much more important than static 'uses' and 'partof' relations. One could ask how important the static 'uses' and 'partof' relations are when compared with:

- dynamic aspects and contention for shared resources,
- technology choices (e.g. for message passing),
- choice of algorithms (e.g. for scheduling),
- error handling,
- diversity and product families,
- choice of commercially available building blocks (e.g. kernel, DMBS),
- evaluation of alternative architectures (e.g. performance).

There is no doubt that these issues are important, but if one looses control over the complexity of the actual code, and if the nice architecture descriptions are not related to the real systems, one has a fundamental problem which will hinder most efforts to address dynamic aspects, algorithms etc.

Moreover, a significant part of the complexity of e.g. distributed object-oriented applications consist of all the objects having attributes, most of which are nothing but pointers to other objects: clients know servers, servers keep track of delivery addresses for active clients, routing tables map addresses to addresses, and so on. These pointers and address tables can be summarised as relations too. There certainly is a need for the analysis of these dynamic relations, and maybe the same technology of calculating with relations can be used (provided we can extract relations dynamically from the running system). Continuing this line of though we could find that visualisation software and the relational calculator become software components which can be linked and loaded as a part of the system whose architecture is monitored.

8 References

[1] G.C. Murphy, D. Notkin, K. Sullivan. *Reflecting source code relations in higher-level modules of software systems.* Technical report 94-09-03 Department of computer science and engineering, University of Washington,(1994). (or see http://www.cs/washington.edu/research/tr/techreports.html).

[2] D.R. Harris, H.B. Reubenstein, A.S. Yeh. *Reverse Engineering to the architectural level.* ACM 0-89791-708-1/95/0004 (1995).

[3] G. Schmidt, T. Ströhlein. *Relations and Graphs*, Springer-Verlag (1993).

[4] I. Carmichael, V. Tzerpos, R.C. Holt. *Design maintenance: unexpected architectural interactions.* IEEE, ICSM (1995)

Automatic Synthesis of SDL from MSC in Forward and Reverse Engineering

Nikolai Mansurov

Department for CASE tools
Institute for System Programming, Moscow Russia
25 B. Kommunisticheskaya,
Moscow, Russia
Email: nick@ispras.ru

Abstract

In this paper we present Message Sequence Charts (MSC) as a formal technique applicable both for very early and very late phases of the software development process. We demonstrate how the synthesis technique, producing executable specifications in telecommunications standard Specifications and Description Language (SDL) from MSC can be used to support early phases of software development in a forward engineering process as well as integration with older, legacy software as a reverse engineering technique. The methodology presented in this paper is aimed at lowering the "barriers" for wider adoption of formal specification languages in industrial context.

1. Introduction

CASE-based approaches offer significant improvements in quality, productivity, and time to market [5]. However there exist certain *barriers* for wider adoption of formal methods in industry. We identify two major barriers – support of *early development phases* [4] and support for integration with *legacy* software [2].

There exists a significant gap between mathematical-based formal methods and design practice at the early phases of the software development process [8]. The design of a new system usually starts in an exploratory and iterative way with a Requirements Capture phase. At this phase the problem domain is surveyed, and fragments of a trial solution are sketched. Most of these sketches lead a short life, and are modified frequently. In the initial phases of a design, comprehensive formal specification and verification techniques offer little help to the designer [8]. They appear to require a level of formality and precision that is not available yet. In return, only fairly abstract properties may be established. The initial price to be paid is too high, the initial rewards are far too small [8].

Instead, the so-called use case based methodologies are becoming predominant in software development [9,16]. Use case based methodologies share the common way of capturing the customer requirements as *scenarios*. Message Sequence Charts (MSC) [7] or Sequence Diagrams of the Unified Modeling Language (UML) [16] can be used to model use cases. The MSC language is especially attractive as an FDT for the early phases of the software development process because it is well accepted in the telecommunications industry and has a low learning curve and, while at the same time it has a well-defined formal semantics. We believe that significant improvements of the time-to-market can be gained by expanding the use of FDT-based CASE tools to the early phases of the software development process [5,4,8].

Apart from the support for the early design phases, there is another important issue which needs to be addressed in order for formal methods-based CASE tools for communications software engineering to become common practice. Formal methodologies are only applicable to the so-called "*green-field*" projects, in which the system is developed completely from scratch. However, most projects in the industrial context involve the older, "*legacy*" base software. This software is being maintained, updated by developing new features, or reused in new projects. For the formal methods to be adopted in industry, it is necessary to provide cost-effective methods for integrating CASE-produced components and systems with older, "legacy" base software [1]. Legacy software systems were produced with older development methods, often involving a blend of higher-level code, and system-level code, with heterogeneous languages, architectures, and styles, and often very poorly documented. Up to now, this fact has constituted a "*legacy barrier*" to the cost effective use of formal methods-based development technologies and tools [2]. In order to overcome the "legacy barrier", there is an increasing demand for developing cost-efficient re-engineering methods which will significantly reduce the effort involved in creating formal specifications of the base software platforms.

In this paper we describe our experience in lowering the barriers for wider adoption of formal methods in telecommunication industry. We have selected MSC as the "interface" formal method, intended for use by humans. We developed synthesis methodology aimed at producing executable specification in another telecommunications standard formal language called Specification and Description Language (SDL) [6]. We discuss methodological issues of using MSCs and synthesized SDL specifications at both early and later phases of the development process.

The rest of the paper has the following organization. Section 2 presents our synthesis technique, including data extensions to MSC language and the key concepts of the synthesis algorithm. Section 3 describes application of synthesis for forward engineering phases of the software development. We describe high-yield requirements validation and architecture validation technique. We also provide comparison to several related approaches for using synthesis as part of forward engineering process. In Section 4 we describe applications of synthesis for reverse engineering. We present our dynamic scenario-based approach to re-engineering of formal SDL specifications of legacy telecommunications software. We also provide comparison to some related approaches for re-engineering formal specifications. Section 5 concludes our presentation.

2. Synthesis of Executable SDL Specifications from MSC

In this section we present our technique of technique for synthesis of SDL models from scenarios formalized in MSC. We describe formalization of scenarios, our data extensions to MSC, describe the key concepts of the synthesis algorithm, and provide an illustrative example.

2.1. Overview

Our approach to formalization includes the following guidelines:

- Each scenario is formalized using *a Basic Message Sequence Chart* (bMSC) [15]
- Control-flow relationships between scenarios are formalized using *High-Level MSCs* (HMSC) [7]. Control-flow relationships between scenarios include alternative (sub-) scenarios, iterations of (sub-) scenarios, "uses" and "extends" relations between scenarios [16]

- Data-flow relationships between scenarios are formalized using our *data extensions* [4] to MSC language
- Certain *composition rule* is used [4]:
  - Sequential rule (single scenario is executed “to completion”)
  - Parallel rule (different scenarios can execute simultaneously)
  - Multiple instance rule (multiple instances of the same scenario can execute simultaneously)
- MSC instances are mapped to finite state automata extended with data operations
- SDL models are automatically synthesized, such that they are:
  - complete (both structure and behavior) & ready to run
  - non-deterministic
  - typebased

2.2. Data Extensions to MSC

Let’s consider our data extensions [4] in more detail.

1. **Variable definitions**. We allow to define variables of different types. SDL semantics is assumed. Variable definition is placed into a text symbol in any MSC diagram. A local copy of each variable is propagated to each actor. Simplified SDL syntax is used for variable definitions:

```
<variable definition> ::= dcl <var_name> <type>;
```

2. **Actions**. We allow MSC action symbols to contain operations on local variables. SDL semantics is assumed. Simplified SDL syntax is used for actions:

```
<assignment> ::=    <var_name> := <expr>
<function call> ::= <func> (<expr1>,…,<exprn> )
```

3. **Message parameters**. We allow messages to have parameters. We restrict the syntax of message parameters to variable names. SDL semantics is assumed for parameter passing.
4. **Create parameters**. Actors are allowed to have parameters which are passed from the parent instance to the child instance during the create event. We restrict the syntax of create parameters to variable names. SDL semantics is assumed.
5. **Local conditions**. We allow to specify local decisions using boolean expressions over instance variables. Syntactically, local decisions are specified as local conditions on the axis of the corresponding instance. The boolean expression is written in a comment box attached to the local condition. Semantics of such condition is that the subsequent events are considered only when the value of the boolean expression is true. Boolean expressions are restricted to the following syntax:

```
<boolean expression> ::= <var_name> <op> <var_name>
                         <var_name> <op> <const>
```

Alternative sequences of events can be specified in a different MSC using a local condition with the same name and a different guard. All guards must be mutually exclusive.

6. **Timers**. Subsequent *set* and *timeout* events on an MSC instance axis may be used to specify a delay during use case execution. In an abnormal scenario such delay may specify an expired timeout which causes an error. Note that timers with parameters are not supported.

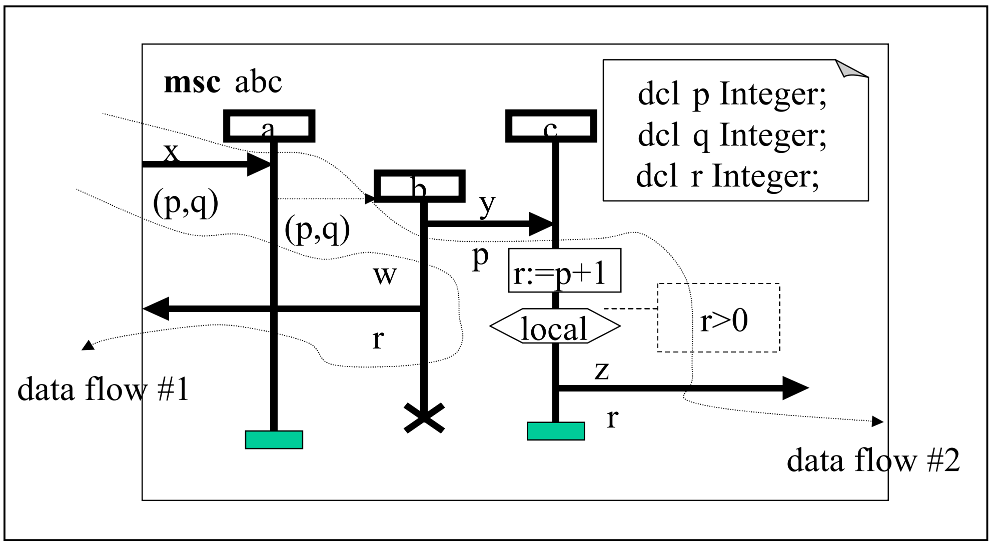


Figure 1

Our extensions to the MSC language describe the *flows of data* through individual scenarios (local data flows) as well as data flow dependencies between scenarios (global data flows). The concept of data flows over scenarios is illustrated in Figure 1 and Figure 2. Two local data flows through scenario *abc* are shown (as two dashed lines) in Figure 1. The first flow contains the following MSC events: a: **in** x(p,q) **from env**; a: **create** b(p,q); b: **out** w(r) **to env**; Note, that instances *B* and *C* use different (local) copies of the variable r. Thus the parameter of the message w which is sent by the instance *B* to the environment is not necessarily equal to p+1.

The second flow contains the following MSC events: a: **in** x(p,q) **from env**; a: **create** b(p,q); b: **out** y(p) **to** c; c: **in** y(p) **from** b; c: **action** 'r:=p+1'; c: **out** z(r) **to env**;

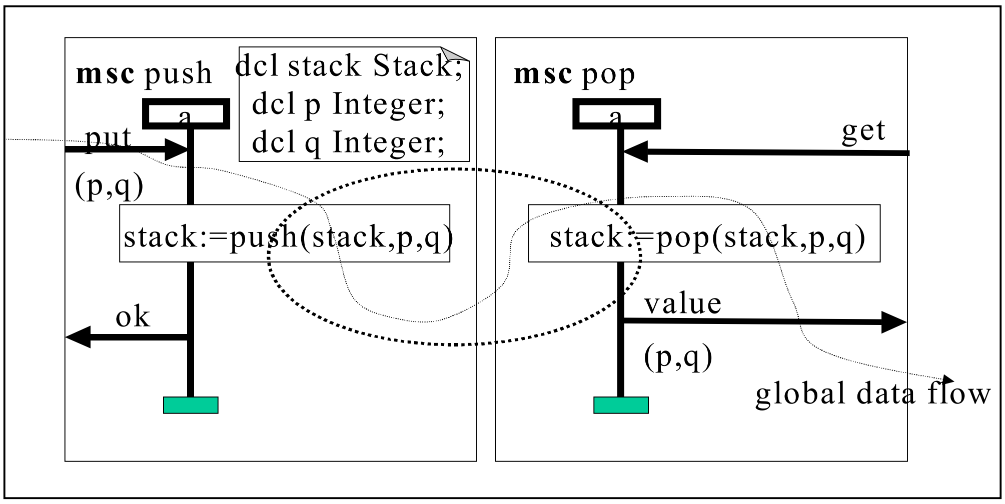


Figure 2

Note that the instance *C* will send message z(r) to environment only when condition r>0 is true. Alternative events for the instance *C* can be specified using the local condition local with a different guard. Global conditions in the HMSC graph will be required when alternative events involve other instances.

Figure 2 illustrates the specification of global data flows between use cases. In Figure 2 parameters of the message *value* returned by the use case *pop* as a reaction to the message *get* depends on the events in the use case *push*. `push` and `pop` operators are assumed to be user-defined SDL procedures with in/out parameters implementing well-known stack operations.

Our main motivation in adding data extensions to MSC is to allow more accurate specifications of functional requirements. However the same data sub-language turns extended MSC into a powerful FDT for design phases.

2.3. Synthesis Algorithm

We use the concept of *event automata*. An **event automaton** is a finite automaton corresponding to *a single MSC instance*, such that the input symbols for the automaton are MSC events, involving the given MSC instance. We distinguish between three categories of MSC events: input, active and idle events. An idle event is a trivial (empty) event, which was added to simplify algorithm description.

- input events (require synchronization with other instances, decision about event is taken by another instance)
  - message input *in(i,m)*
  - timeout *timeout(t)*
- active events (do not require synchronization with other instances, decision is local to the current instance)
  - message output *out(i,m)*
  - action *action(a)*
  - set timer *set(t,d)*
  - reset timer *reset(t)*
  - stop action *stop*
  - local condition *check(c)*

Our synthesis algorithm constructs a particular kind of event automata, which we call *MSC slices*. An **MSC slice** (corresponding to an MSC instance *i)* is an event automaton, producing all valid event traces for the instance *i*. We use the following algorithm to construct MSC slices:

1. initial states of the event automaton correspond to symbols at HMSC graph with idle events
2. for each bMSC a (sub)sequence of states is created, corresponding to the sequence of events involving the instance *i*
3. each MSC reference is replaced by the corresponding (sub)sequence of states
4. the start symbol of the event automaton corresponds to the HMSC start symbol

Our synthesis algorithm consists of the following steps [4]:

1. Integrate HMSC model
2. Construct *MSC slices*
3. Make MSC slices deterministic
4. Minimize MSC slices

5. Generate SDL behavior
6. Generate SDL structure

2.3. Example

Let's consider a simple MSC model shown in Figure 3. It contains two use cases *Wait* and *Reply* (each of them has only one scenario). We assume sequential composition rule. Instance *R* (*receiver*) corresponds to the system actor and instance *S* (*sender*) is an external actor.

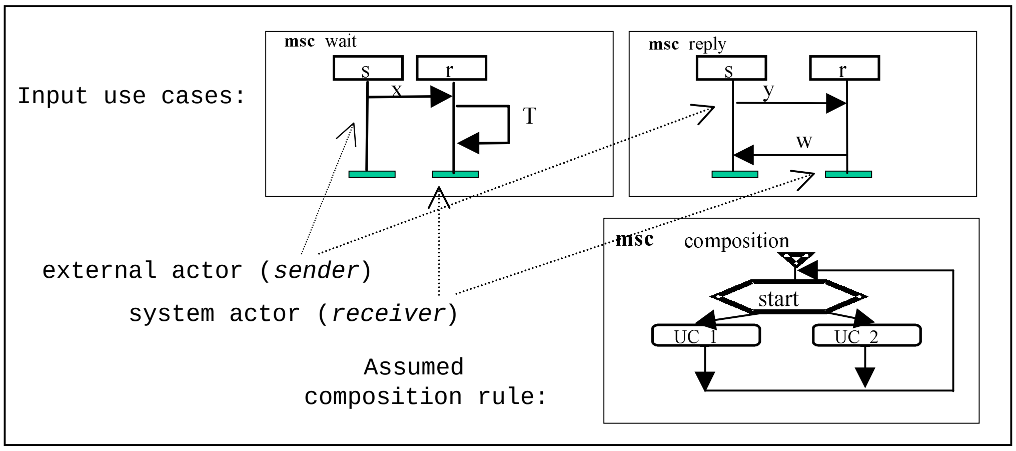


Figure 3

Sender S initiates both use cases. Use case wait is started by sender S sending message X. Receiver R has to wait for an unspecified period of time. Use case reply is started by sender S sending message Y. Receiver R has to respond with message W.

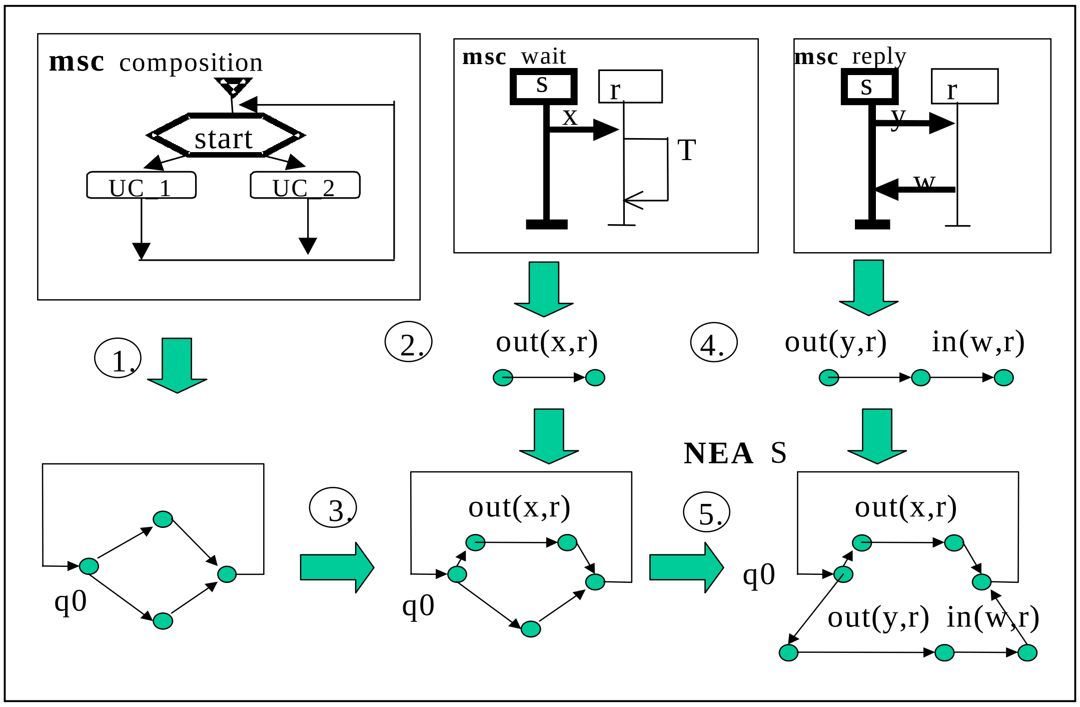


Figure 4

Figure 4 illustrates our algorithm for constructing an MSC slice for instance S. The first step of the algorithm constructs an event automaton from the HMSC graph (step 1). Then an event automaton is constructed from the instance S at MSC wait (step 2). This automaton has only one transition, labeled with an active event *out(x,r)*. Then the corresponding transition in the initial event automaton is substituted for the newly created event automaton for instance S from MSC wait (step 3). The following two steps (steps 4 and 5) process instance S from MSC reply. The resulting MSC slice is presented in the bottom right corner of Figure 4. It is labeled non-deterministic event automaton (NEA).

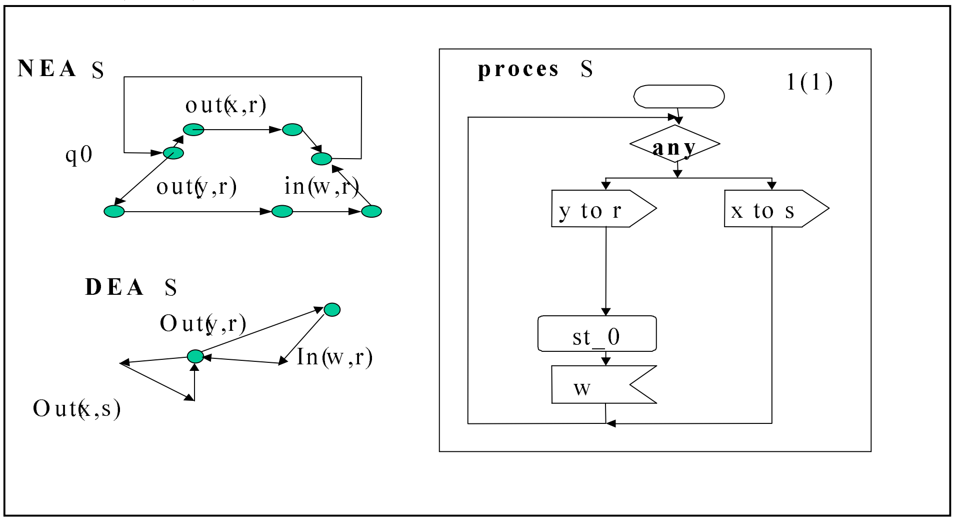


Figure 5

Figure 5 illustrates subsequent steps of our synthesis process. The non-deterministic MSC slice for instance S (NEA S) is made deterministic (in the event automata sense) and minimized (DEA S). An SDL graph is then generated (process S). Figure 6 illustrates the generated SDL structure for our example.

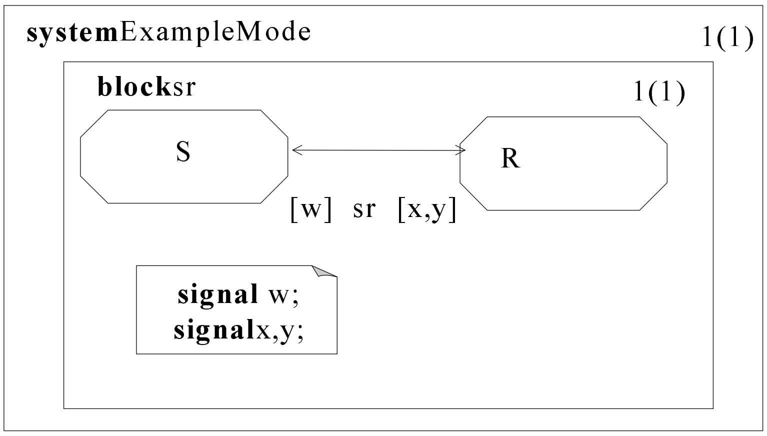


Figure 6

3. Applications for Forward Engineering

3.1. High Yield Requirements Validation

In this section we describe the use of synthesized SDL models for requirements validation. Requirements validation is used to detect faults in the customer requirements [5]. Requirements validation is a form of testing applied to an early phase. Requirements validation is an iterative process consisting of the following steps (see Figure 7):

1. Formalize requirements in the form of use case scenarios
2. Synthesize executable requirements model from scenarios
3. Create validation scenarios
4. Run validation scenarios through the requirements model
5. Validate the execution sequence of each validation scenario to either
   - 5.1. Accept the validation scenario. In this case the validation scenario can be included into requirements use cases
   - 5.2. Reject the validation scenario. In this case the initial customer requirements contain a fault. E.g. the initial requirements can be inconsistent or incomplete. The rejected validation scenario has to be transformed into a use case and the initial requirements need to be updated by including the new use case and removing any existing inconsistencies.
6. Check termination criteria and start with a new iteration, if necessary (from step 2).

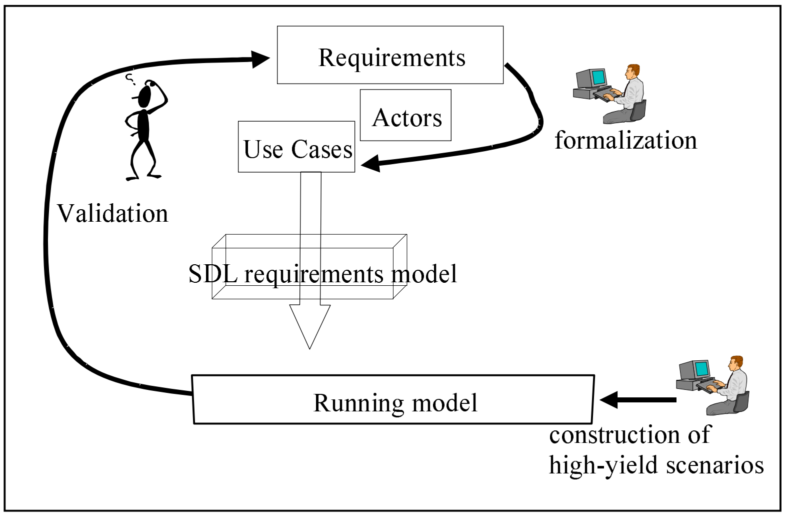


Figure 7

High-yield requirements validation is a risk-based approach, being developed by Prof. R. Probert at the Telecommunications Software Engineering Research Group (TSERG) of the University of Ottawa [5]. Same considerations are applicable to requirements validation as to testing. One cannot test a program to guarantee that it is error-free. Since exhaustive testing is out of the question, we must maximize *yield* on the testing investment (i.e., maximize the number of errors found by a finite number of test cases *Yield* of a validation scenario refers to the number of defects detected at by a particular scenario. An alternative definition of yield can refer to the amount of risk removed by the scenario. *Risk* refers to the "cost" associated with a failure. By definition, Risk R = PF * CF (where PF is the probability of failure; CF is the cost of failure).

Let us introduce some terminology for discussing *validation scenarios*. We make a distinction between *primary scenarios* (normal, everything works as expected, success paths) and *secondary scenarios* (alternative, exceptional, race conditions, collisions, known pathological sequences of client/system interactions, fail paths). All *functional scenarios* (scenarios which describe how a user achieves a particular service or capability) are primary, scenarios which describe how he/she was thwarted are secondary. In general, scenarios which are essential and desired by a customer are primary. Primary scenarios are denoted "*low-yield*" since they describe situations and actions which are generally well understood. The yield (detected or anticipated error count) is therefore low. Secondary scenarios, on the other hand are denoted *moderate* or *high-yield*, since they describe situations and interactions which are generally not well documented, and therefore are not well understood. The associated yield for such scenarios is high because designer choices are likely to differ from client choices, or to be non-deterministic.

The objective of the high-yield requirements validation is to focus the effort on the elements with highest risk. *Low-yield* scenario is not likely to detect a defect by causing an observable failure because such scenarios are in general well-understood by both the customers and the developers. On the other hand, *high-yield* scenarios have a high probability of detecting a defect. High-yield scenarios correspond to the secondary scenarios, e.g. exceptional behavior (error-handling behavior path). Usually, these scenarios are less well understood by the developers.

In [4] we suggested to automatically synthesize an SDL requirements model at the requirements analysis phase (Figure 7). Automatic synthesis of SDL models from MSC has the following benefits:

- MSC modeling allows high-yield requirements validation by *simulation* of SDL models using high-yield scenarios
- MSC models can be developed concurrently while architecture integrity can still be maintained via iterative synthesis
- Regression testing is eliminated because accepted validation scenarios are added to the set of validation scenarios and the synthesized model is by construction correct with respect to the previously accepted behavior
- Early fault detection can be performed by the synthesizer
- Different compositions of use cases can be explored (single, concurrent, etc.)
- Slices of the MSC model can be created, explored and reused

The *synthesized requirements model* (SRM) can be used to generate additional scenarios which are *longer* than the original validation and therefore provide better understanding of the requirements [14]. Simulation of the synthesized requirements model allows to quickly discover inconsistencies and incompleteness of the requirements because the synthesized model will generate many variations of the original scenarios, including abnormal behavior. Such scenarios are likely to be less well understood by the developers.

3.2. Architecture Validation

Automatic synthesis can also be used at the system analysis phase. During this phase the *architecture* of the system is being defined and independent groups of developers produce system scenarios for each architecture component (Figure 8). The input at this phase is a set of system scenarios. Normally a *system scenario* will be a refinement of the corresponding scenario from the requirements analysis phase, capturing the interaction of the architecture components. The structural information available in the system scenarios consists of the set of external actors and the architecture components (represented as distinct instances in the MSC model). The behavioral information is available in the form of functional scenarios representing the typical interactions between the architectural components as well as between external actors and the architectural components. Additional behavioral information can be captured in the form of the data flows over the system scenarios. The automatically synthesized model created at this phase is called the *synthesized architecture model* (SAM). The feedbacks provided by SDL tools through the SAM go both to the system scenarios as well as to the architecture model (Figure 8).

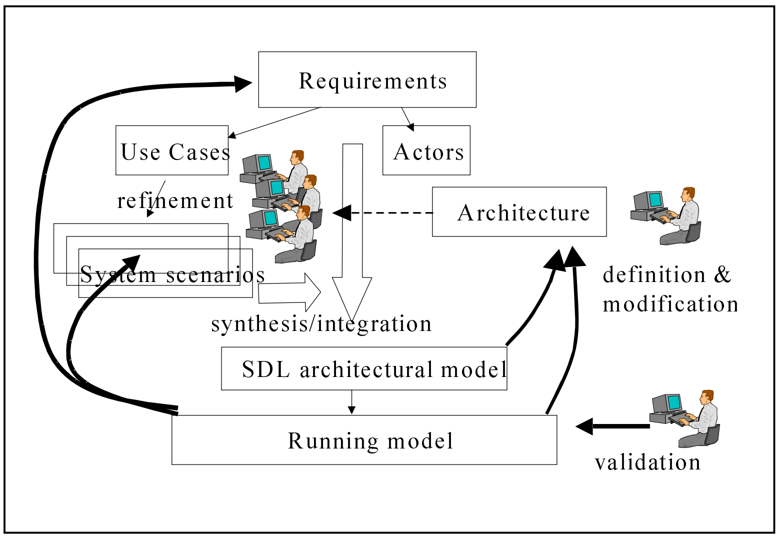


Figure 8

In this case the SAM will reproduce the architectural components of the system by deriving them from the collection of system scenarios, synthesize the behavior of each component and integrate the model. Automatically derived relationships between components can be compared to the intended ones (described in the architecture model). In our experience the synthesized architecture model is helpful in uncovering system analysis faults. This step is a direct continuation of requirements validation activity described earlier.

Vertical decomposition of MSC models can be used in conjunction with our synthesis technique to refine requirements models into architecture models. Figure 9 demonstrates this approach. In use case reply from example at Figure 3, instance R is decomposed into three instances: R1, R2, R3. Message flow between decomposition instances has to be compatible with the message flow at the parent MSC diagram. Additionally, two alternative behavior paths are now specified for the reply use case. Alternatives are specified using MSC condition with same name. The synthesized architectural SDL model is presented at Figure 10.

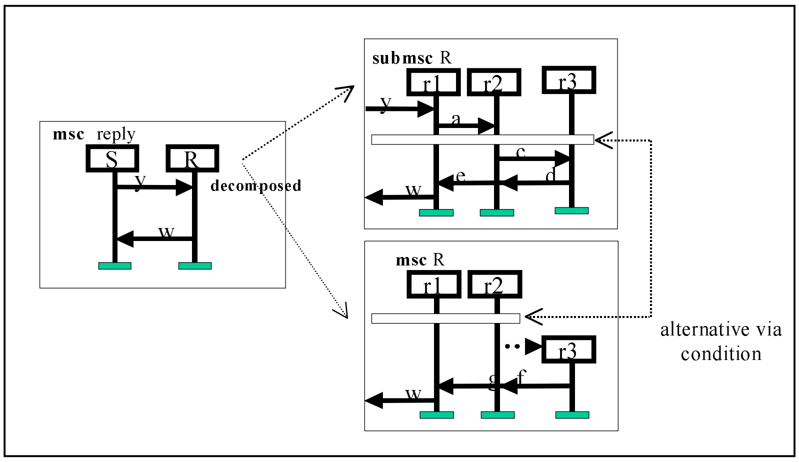


Figure 9

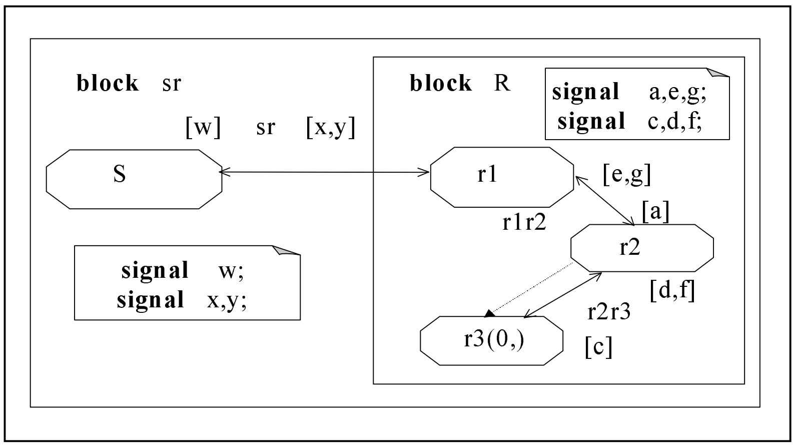


Figure 10

3.3. Comparison to related work

Automatic synthesis of executable models from scenarios is an active research field. Much work has been done on the subject of translating MSCs to other languages [14,10]. Synthesizing SDL specifications from MSC is addressed in [10]. Survey of work on a more general subject of protocol synthesis is available in [13].

Methodological issues of generating a formal executable specification from a set of use cases are addressed in [11]. This paper summarizes experience in manually developing a LOTOS specification of a telecommunications standard on the basis of use cases provided by industry. LOTOS tools were used to validate the specification and generate all original use cases as well as additional ones. The main motivation of the project was to use LOTOS tools to analyze and maintain a set of use cases. The benefit of using the formal executable specification for prototyping purposes was emphasized.

The University of Montreal synthesizer [15] translates scenarios with timing constraints into timed automata. The main motivation of the project is to provide formalization of scenarios and ensure the accuracy of requirements analysis.

The Waterloo synthesizer [14] translates MSC models into ROOM specifications. The main motivation of this project is to create an executable architectural model supporting design phases. Firstly, an executable architecture model was considered useful for prototyping purposes. Synthesized ROOM models can be simulated by ObjecTime Developer tool with the possibility to visualize execution sequences as bMSCs. According to [14], the MSC traces are useful for visualizing execution sequences that are longer that the bMSC scenarios in the original MSC specification and therefore provide a better overview and understanding of the system. Executable architecture models were considered helpful in supporting communication and education of new team members. Secondly, automatic synthesis of architectural models was considered useful in evolutionary prototyping by providing refinements to the model. Designers can modify the synthesized model, execute a number of scenarios, and then feed the results back into the domain of MSC specifications. The possibility of ObjecTime Developer to automatically generate C++ code skeletons was also considered beneficial.

The motivation of the Moscow synthesizer is similar, however we also use automatic synthesis to create executable requirements models. We decided to use SDL as the target language because of the better tool support available for SDL.

The Waterloo synthesizer produces architectural models with both structural and behavioral components [14]. The Waterloo synthesizer derives static process structure based on the instances in bMSC. Similar approach is taken in the Moscow synthesizer. Additionally, the Moscow synthesizer derives *dynamic process structure* by considering bMSC with instance creation and deletion. When synthesizing behavior components, the Waterloo synthesizer considers only message input and output events. The Moscow synthesizer additionally considers timer events and supports *data flow extensions* to the MSC language (variables, message and create parameters, actions and local conditions with guards).

The Concordia University synthesizer [10] translates MSC models into SDL specifications. The main motivation of the project is to eliminate validation of SDL specifications against the set of MSCs by ensuring consistency between the SDL specification and the MSC specification through automatic synthesis [10]. The main characteristic of the Concordia synthesizer is that the architecture of the target SDL specification is required as an input to the synthesis algorithm and the question of implementability of the given set of MSCs within the given SDL architecture is addressed [10]. Thus the Concordia synthesizer produces only behavioral components. Composition of bMSCs using HMSC was not addressed in [10] although it was considered as a direction for future work.

Although the Moscow synthesizer was developed independently, some of the technical decisions are similar, e.g. the use of SDL save statement to avoid deadlocks in the synthesized SDL models. However the motivation of the Moscow synthesizer is somewhat different. The Moscow synthesizer produces both the behavioral and the structural components (similar to [14]) which allows us to synthesize executable requirements models (similar to [11]) as well as executable architecture models [14]. Consideration of data flows in the Moscow synthesizer allows more accurate capture of the functional requirements as well as more accurate capture of the architectural issues.

4. Applications for Reverse Engineering

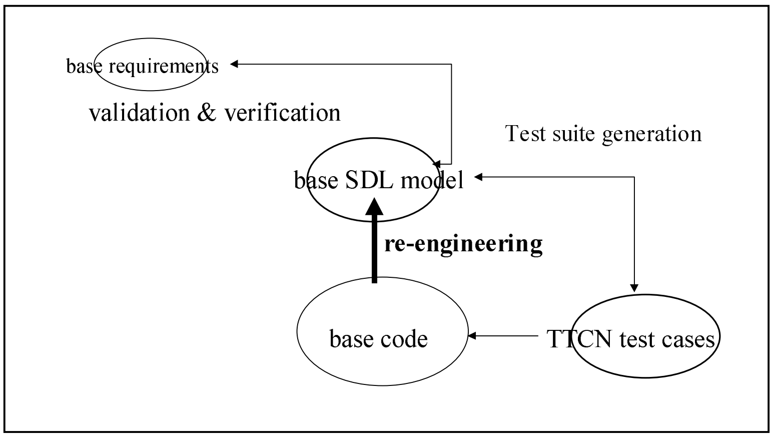


Figure 11

Legacy software systems were produced with older development methods, often involving a blend of higher-level code, and system-level code, with heterogeneous languages, architectures, and styles, and often very poorly documented. Up to now, this fact has constituted a "*legacy barrier*" to the cost effective use of new development technologies [5,2].

In order to overcome the "legacy barrier", there is an increasing demand for developing automatic (or semi-automatic) re-engineering methods which will significantly reduce the effort involved in creating formal specifications of the base software platforms. Cost-effective methods for producing SDL models of the base software platform will allow the following benefits (Figure 11):

- better understanding of the operation of the legacy software through dynamic simulation of the SDL model, which often produces more intuitive results and does not involve the costly use of the target hardware;
- automated generation of regression test cases for the base software platform;

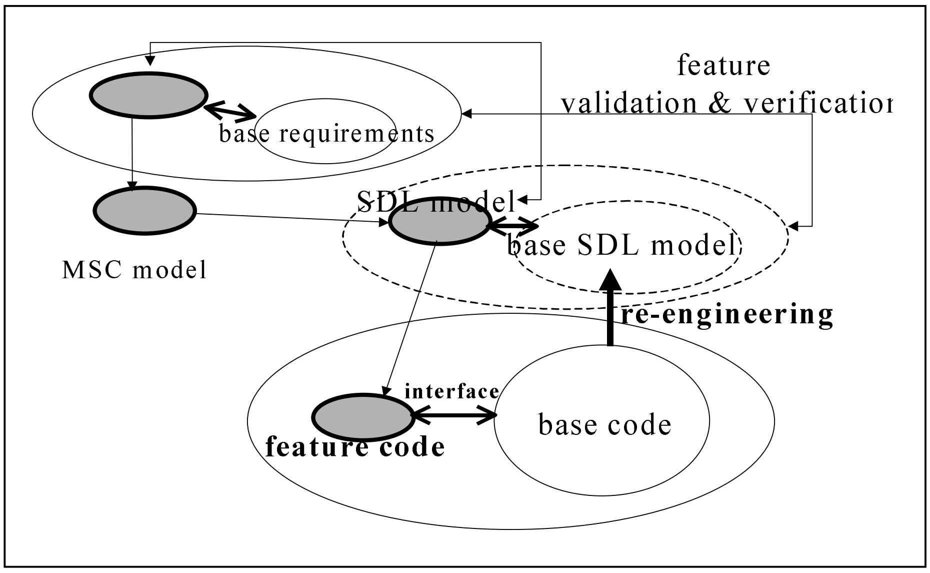


Figure 12

Additional benefits can be obtained for using formal methods *for new feature development* (Figure 12):

- analysis and validation of the formal specifications of the new features built on top of the SDL model of the base software platform;
- feature interaction analysis including existing and new features;
- automated generation of test cases for new features;
- automatic generation of implementations of the new features.

4.1. Dynamic scenario-based approach to re-engineering

In this section we describe our methodology of *dynamic scenario-based* re-engineering of legacy telecommunications systems into a system design model expressed in SDL [2].

Our approach consists of

- placing semantic probes [2] into the legacy code at strategic locations based on structural analysis of the code,
- selecting key representative scenarios from the regression test database and other sources,
- executing the scenarios by the legacy code to generate probe sequences, which are then converted to MSCs with conditions and
- synthesizing an SDL-92 model from this set of Message Sequence Charts (MSCs) using the Moscow Synthesizer Tool [4].

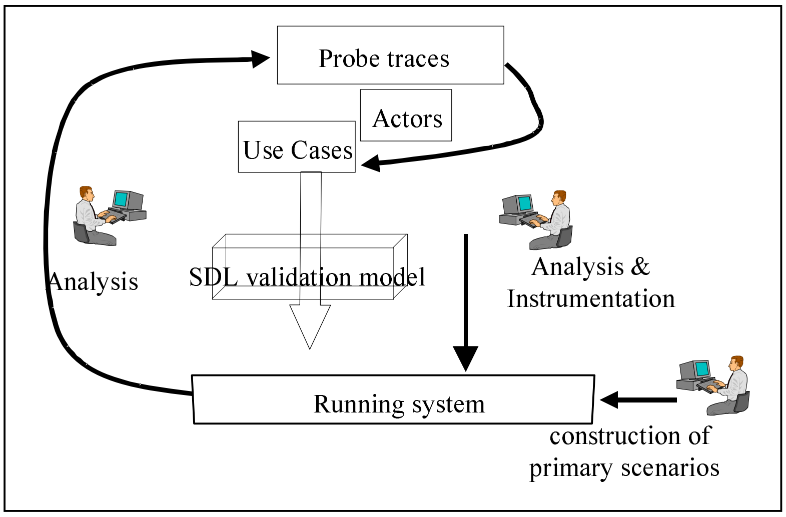


Figure 13

This process is repeated until the SDL design model satisfies certain validity constraints [2]. This SDL model is then used to assess and improve the quality and coverage of legacy system tests, including regression tests. The approach may be used to re-engineer and re-test legacy code from a black-box (environment), white-box (source code), or grey-box (collaborations among subsystems) point of view [2].

4.1.1. Overview

Dynamic scenario-based re-engineering of legacy software into SDL models is a process, where an SDL model is synthesized from *probe traces* [2], collected from *dynamically* executing the *instrumented* legacy system (see Figure 13,14). More specifically, in the process of scenario-based re-engineering, the SDL model is synthesized from a higher-level representation - *extended MSC-92 model (*later referred to simply as MSC model) which is abstracted from probe traces. The execution is driven by a *test suite* [2].

The enabling technology for our dynamic scenario-based re-engineering process is automatic synthesis of SDL models from a set of MSCs [4]. So far automatic synthesis of SDL models from MSC was considered only as a forward engineering technology (see Section 3). In our dynamic scenario-based re-engineering process we exploit the duality of MSCs as both a requirements capturing language and a trace description language which allows us to treat probe traces as requirements for the SDL model.

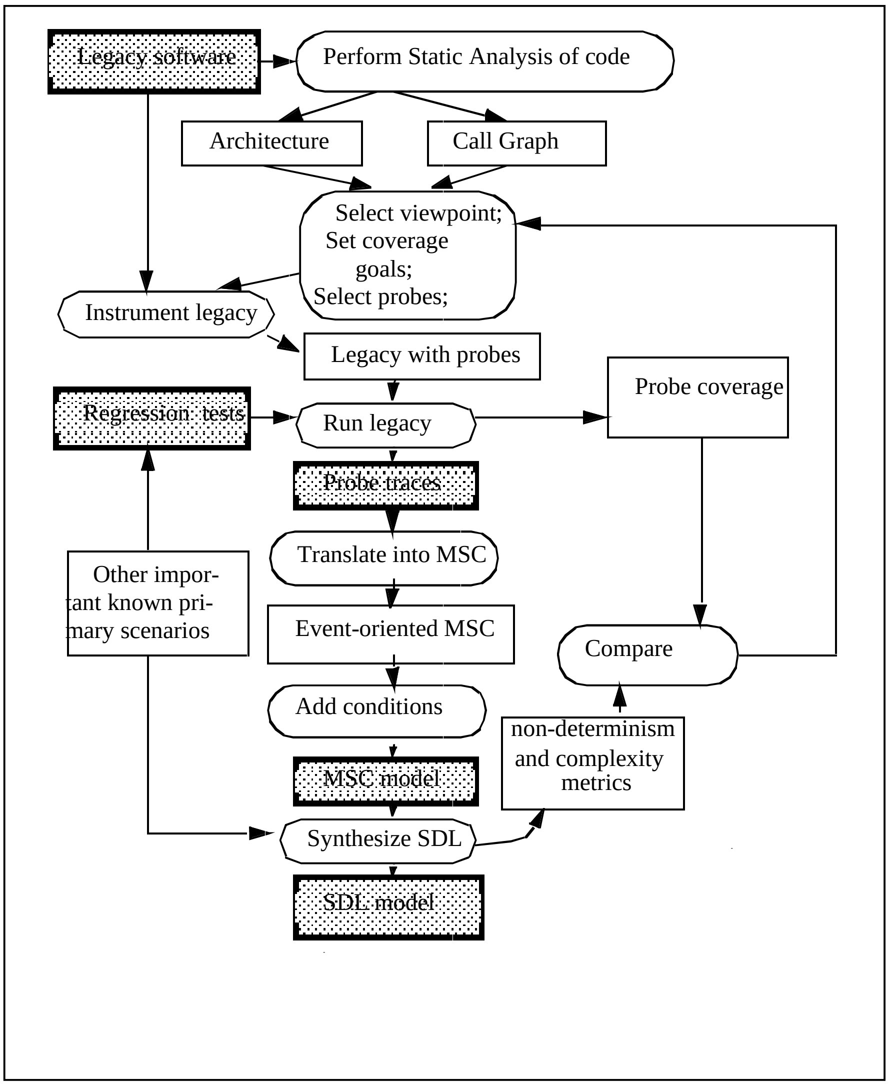


Figure 14

Our re-engineering methodology is an *iterative* process, consisting of the following four *phases*.

1. Preparation
2. Dynamic collection of probe traces
3. Synthesis of SDL model
4. Investigation of the SDL model

Each phase involves a few *steps.* Iterations are controlled by *validity criteria*, which are checked during the last phase. An overview of all steps of the methodology is shown in Figure 13,14. In Figure 14 the methodology is presented as a dataflow diagram. Important *artifacts* are represented as rectangles; methodology steps (sub-processes) are represented by ovals. The main artifacts of our re-engineering process are highlighted. Lines in Figure 14 represent flows of data, which determine the sequence of methodology steps. A detailed description of methodology steps is contained in the next section.

4.1.2. Preparation phase

This aim of this phase is to develop a *probe placement strategy* and select the set of scenarios which will drive execution of the instrumented system and resulting probe trace capture.

Step 1. **Analyze code.** This step uses well-known methods of static structural analysis to select probe placements. Two models of software can be used as guidelines for probe placement - the *architectural model* of the system (major components and their relationships) and the *call graph* of the system [3]. The call graph of the system should identify *external interfaces* of the system (usually - system calls of the target operating system, or assembly inline code).

Step 2. **Select modeling viewpoint**. Our approach may be used to re-engineer and re-test legacy code from a *black-box* (environment), *white-box* (core code), or *grey-box* (collaborations among subsystems) point of view. Viewpoint determines the structure of the resulting SDL model.

Step 3. **Set coverage goal and select probes**. At this step we finalize probe placement by selecting particular locations in the source code of the system where probes are to be placed, and defining the format of the information generated by each probe. By selecting the coverage goal we control the level of details in traces and thus determine the external interface of the model. The external interface of the model is determined in terms of locations on the architectural model of the system and the call graph, such that probes register desired events and collect desired data.

Semantic probing [2] is assumed. Coverage requirement is not phrased in terms of syntactic entities such as statements or branches, but in terms of semantic entities, namely *equivalence classes* of program behavior [2]. These equivalence classes of program behavior are determined solely from the system design. Probe traces obtained by executing instrumented code can be related directly to the system design. Inspection of probe traces may drive modification of semantic probes and thus lead to further iterations of the re-engineering process.

Step 4. **Collect known primary scenarios + regression tests.** The dynamic capture of probe traces is driven by the *test suite*. We suggest that the (legacy) regression test suite be used to drive the first iteration of scenario-based methodology.

We start our iterative re-engineering process with regression tests. *Regression tests* consist of a blend of *conformance tests* (usually success paths and therefore low-yield), primary scenarios (low-yield), and a few known important secondary scenarios (moderate to high yield). We continue with additional functional (primary) scenarios as required to improve the semantic capture of our SDL model. As our iterations converge, we are more interested in secondary higher-yield scenarios. Discussion of the yield of scenarios with respect to requirements validation was presented in Section 3.

4.1.3. Dynamic collection of probe traces

The aim of this phase is to capture the set of probe traces, which correspond to the probe placement strategy and selected scenarios.

Step 5. **Instrument legacy.** Suitable *probing infrastructure* for generation and collection of probe traces needs to be established. Probes need to be inserted into the source code according to the placement strategy.

Step 6. **Run legacy code to generate probe traces.** The legacy system needs to be built and executed on a test suite. The target or simulated environment together with the existing testing infrastructure are used. The result of this step is a collection of *probe traces*. Another result of this step is the measurement of *probe coverage* of the system by the current test suite.

4.1.4. Synthesis of SDL model

This is the key phase in our methodology. The aim of this phase is to synthesize an SDL model of the legacy system.

Step 7. **Translate probe traces into event-oriented MSCs.** This step was introduced into the methodology in order to separate two different concerns - dynamically capturing scenarios from legacy and synthesizing SDL models from scenarios. This step performs a (simple) translation between traces and MSC. This step is determined mostly by the differences between the format of probe traces (as defined at the instrumentation step), and the format of input to the synthesizer tool.

Step 8. **Add conditions to MSCs.** This step was described as “abstraction” in Figure 1. The aim of this step is to identify transaction-like sequences of interactions, corresponding to requirement use cases. Linear MSCs (corresponding to traces) are then converted into an MSC model, which corresponds to requirement use cases. This is done by inserting conditions [7] into places where loops or branching are possible. Note, that we are using an extended event-oriented MSC-92 notation as the input to the MOST-SDL tool [4]. In MSC-96 this corresponds to creating an HMSC.

Adding conditions to MSCs can significantly improve the amount of information, contained in MSCs which will lead to synthesis of models with more interesting behavior.

Step 9. **Synthesize SDL model.** This step is done automatically by applying the Moscow Synthesizer Tool (MOST-SDL). Synthesizer technology is briefly described in the next section. A more detailed description is contained in [4].

The outputs of this step are the 1) synthesized SDL model; and some complexity metrics of the model: 2) number of states in SDL model and 3) *non-determinism metric* of the model. The later metric is an *indirect termination criteria* for the re-engineering process. A non-deterministic choice is generated each time when two or more input scenarios have different behavior on the same external stimulus. In practice this often means that behavior of the system is determined by the previous history, but the traces captured during the previous steps do not contain enough data. High values of the non-determinism metric should lead to further iterations of the re-engineering process.

4.1.5. Investigation of SDL model

The aim of this phase is to check termination criteria by investigating the probe coverage and complexity metrics of the synthesized model, including a very important non-determinism metric.

Step 10. **Terminating criteria.** We need to make sure that the generated model adequately captures the behavior of the legacy system. This may require several iterations of the re-engineering process. Inadequate behavior of the model may be caused by at least two factors: 1) some important primary scenario is not captured in (legacy) regression tests; 2) an abstracted interface of the system is incorrectly selected (missing probe or incorrectly placed probe).

A probe can be incorrectly placed when it a) does not correspond to a desired behavior equivalence class (e.g. two different probes are placed in the same equivalence class); b) probe is placed into correct behavior equivalence class, but is placed in an incorrect syntactical place - into a code location which is not executed when at least some locations of the desired behavior class are executed (e.g. probe is placed into only one branch of a conditional statement).

In our experience, incorrectly placed probes result in errors in probe coverage. Missed probes on input interfaces result in high values of the model non-determinism metric. Missed probes on output interfaces result in errors in generated test coverage. Thus when the probe coverage, non-determinism metric and generated test coverage together are satisfactory the iterations can be terminated.

4.2. Comparison to related approaches

In this section we compare our dynamic scenario-based approach to the so-called direct re-engineering [1] and the so-called partial re-engineering [17]. Schematic representation of the transformations performed by these approaches is shown in Figure 15.

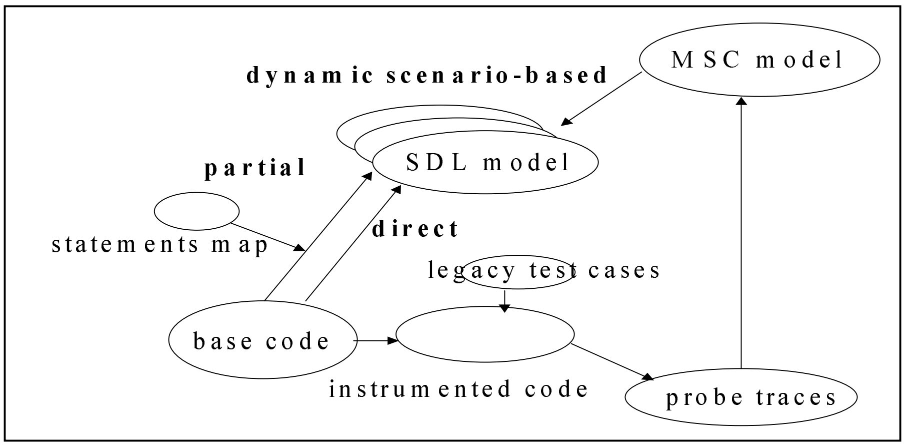


Figure 15

Direct re-engineering approach derives SDL model statically from the source code by performing semantic-preserving translation [1]. Thus the direct SDL model contains *at least the same* amount of information as the implementation itself. In fact, directly generated SDL models contain on average *8-12 times more* information than the implementation, because the mapping from a conventional language to SDL is *divergent*, as demonstrated in [1]. In contrast, SDL models which are synthesized according to our dynamic scenario-based approach always contains *less* information than the implementation.

The so-called *partial re-engineering* [17] is another static approach. It provides an interesting alternative to direct re-engineering. According to this approach, only the framework of the model is extracted automatically (in [17] a state machine model was extracted from a program in C). Extraction of any details of the legacy is controlled by the so-called statements map. The statements map contains all different source statements (in some canonical form) and their translation into the model statements within the automatically generated framework. The statements map is inspected and filled-in manually. By default, the source statements are simply skipped, thus resulting in quite abstract models. Thus the statements map controls the precision of the extracted model. The statements map is relatively stable to the changes in the source code, which makes this approach suitable for evolutionary re-engineering.

Static approaches have certain advantages over dynamic ones since they are independent of (legacy) regression tests, and they usually easier to achieve complete semantic coverage of the legacy. Another important advantage is that static approaches are independent of the target platform. Static re-engineering techniques in general require considerably deeper analysis of source code, and thus are much more expensive. The disadvantage of the direct mapping is that it has to handle large volumes of base software platform source code, therefore - SDL tools need to handle larger SDL models. Partial re-engineering seems quite promising, since it provides a balance of effort between expensive automatic static analysis and manual modification of the statements map. However, the automatic extraction of the state machine framework can be also quite expensive. In [17] cost-efficient extraction of the finite state machine framework was made possible because of the incidental use of a special notation in the code. This notation was introduced for an unrelated purpose, but made extraction of states fairly trivial [17].

The biggest advantages of dynamic scenario-based approach as compared to direct approach, is the flexibility to produce a broad range of distinct models by varying input scenarios and probe placement strategies. In general, scenario-based approach yields more abstract models, which are free from implementation detail. Thus SDL tools could be easier applied to such models. Both kinds of SDL models are *trace-preserving* with respect to the traces produced by the test suite. However, a directly generated SDL model is capable of producing more traces, than those produced by the original test suite, while a scenario-based SDL model is fully defined by the original test suite. On the other hand, traces produced by two SDL models have different *levels of detail*. Traces produced by directly generated SDL model contain all implementation detail, plus some additional detail, introduced by the mapping [1]. The level of detail of directly generated SDL models can be controlled by selecting external interface of the implementation. Traces, produced by scenario-based SDL model are expected to contain much less detail. As demonstrated above, the level of detail of the scenario-based model is controlled by the *probe placement strategy*.

5. Conclusions

Support for early phases of the development process and support for integration with older, legacy software are, in our opinion, two major barriers for wider adoption of formal methods in industry. Ironically, at the early phases, there is “too little” to formalize, while on the other hand, at the later phases there is often “too much” to formalize. However, early formalization is required because it can enable tool-aided feedback and thus allow rapid iterative development. Requirements for an early formalization technique include ease of use, low learning curve, very quick turn-around cycle, maintainability. It is also beneficial to be able to re-engineer formal models of legacy in a cost-efficient way, because it allows to use formal methods for subsequent development of new features, as well as to use tools for better validation of the base software platform.

The key idea, presented in this paper is that MSC is suitable both for the forward and reverse engineering purposes. As a formal technique for capturing requirements, MSC satisfies all usability criteria for early development phases. On the other hand, MSC are suitable to capture “real” scenarios of legacy through collecting probe traces from suitably instrumented source code.

We presented our approach to synthesizing executable SDL models from scenarios formalized in MSC. As demonstrated in this paper, it turns out that this technique provides adequate support for both forward and reverse engineering.

We have given a broad overview of our accelerated development methodology, based on MSC and SDL, which can be used to significantly improve time-to-market in an industrial software development context. In our experience, the use of this accelerated development methodology combined with the use of SDL tools allows between 20 and 30% speedup in time-to-market for a typical telecommunication system. The use of tools in a related project was found to yield a 20-25% improvement in time-to-market; therefore the estimate above is likely quite conservative.

6. References

[1] N. Mansurov, E. Laskavaya, A. Ragozin, A. Chernov, On one approach to using SDL-92 and MSC for reverse engineering, in Voprosy kibernetiki: System Programming Applications, N. 3, Moscow, 1997 (in Russian)

[2] N. Mansurov, R. Probert, Dynamic scenario-based approach to re-engineering of legacy telecommunication software, in Proc. 9th SDL Forum, Montreal, Canada, June 21-26, 1999, Elsevier Science Publishers B.V. (North-Holland).

[3] N. Rajala, D. Campara, N. Mansurov, inSight Reverse Engineering CASE Tool, in Proc. of the ICSE’99, Los Angeles, USA, 1998.

[4] N. Mansurov, D. Zhukov, Automatic synthesis of SDL models in Use Case Methodology, in Proc. 9th SDL Forum, Montreal, Canada, June 21-26, 1999, Elsevier Science Publishers B.V. (North-Holland).

[5] R. Probert, N. Mansurov, Improving time-to-market using SDL tools and techniques (tutorial), Proc. 9th SDL Forum, Montreal, Canada, June 21-26, 1999.

[6] ITU-T (1993), CCITT Specification and Description Language (SDL), ITU-T, June 1994

[7] Z.120 (1996) CCITT Message Sequence Charts (MSC), ITU-T, June 1992

[8] G. Holzmann, Formal Methods for Early Fault Detection, (invited paper) in 4th Int. School and Symposium on Formal Techniques in Real Time and Fault Tolerant Systems, September 1996, Uppsala, Sweden.

[9] I. Jacobson, M. Christerson., P. Jonsson, G. Overgaard, Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley, Reading, MA, 1992.

[10] G. Robert, F. Khendek, P. Grogono, Deriving an SDL specification with a given architecture from a set of MSCs, in Proc. of the 8-th SDL Forum, Evry, France, 23-26 September, 1997, Elsevier Science Publishers B.V. (North-Holland), pp. 197-212

[11] R.Tuok, L. Logrippo, Formal specification and use case generation for a mobile telephony system, Computer Networks and ISDN Systems, 30 (1998), pp. 1045-1063.

[12] M. Andersson, J. Bergstrand, Formalization of Use Cases with Message Sequence Charts, MSc Thesis, Lund Institute of Technology, May 1995

[13] R. L. Probert, K. Saleh, Synthesis of communication protocols: survey and assessment, IEEE Transactions on Computers, 40(4), pp. 468-475, April 1991

[14] S. Leue, L. Mehrmann, M. Rezai, Synthesizing ROOM Models from Message Sequence Chart Specifications, University of Waterloo, Technical Report 98-06, 1998

[15] S. Some, R. Dssouli, and J. Vaucher, From scenarios to timed automata: Building specifications from user requirements, In Proc. 2nd Asia Pacific Software Engineering Conference, IEEE, December 1995.

[16] J. Rumbaugh, I. Jacobson, G. Booch, The Unified Modeling Language Reference Manual, Addison-Wesley, 1999

[17] G. Holzmann, M.H. Smith, A practical method for the verification of event-driven software, in Proc. ICSE’99, pp.597-607, Los Angeles CA USA, May 1999.

Composing Automata in Graphical Languages for Reactive Systems: from Argos to Mode-Automata[*]

Florence Maraninchi Yann Rémond

VERIMAG[†]– Centre Equation, 2 Av. de Vignate – F38610 GIERES
http://www-verimag.imag.fr/PEOPLE/Florence.Maraninchi

1 Introduction

We are interested in graphical languages for the description of *reactive* systems, mainly in the *synchronous* approach.

The term *reactive* was introduced by A. Pnueli [HP88] to qualify the class of systems like communication protocols, real time process controllers, man/machine interfaces,... Specific problems arise for these systems, because of their intrinsic complexity.

Reactive systems are opposed to *transformational* ones, like compilers, which can be described in an appropriate manner by giving the output as a function of the input. There exist methods that allow to decompose the behaviour of such a system into smaller parts, and there exist languages which support these methods (functional languages, with the composition of function as high level construct, or imperative languages, with the notion of procedure).

For reactive systems, this is not very clear yet what decomposition methods can be applied, and what language constructs can be introduced to support them. Indeed, a reactive system maintains a continuous interaction with an *environment*, and its behaviour is a set of sequences of elementary interactions between the system and the *environment*. Reactive bahaviours are intrinsically parallel, because the system is considered as evolving in parallel with its environment, taking input from it, and sending output to it.

However, a lot of work is being done on the topic of reactive systems, and there exist several languages for the description of such systems, together with programming environments. We are particularly interested in synchronous programming lan-

[*]This work has been partially supported by Esprit Long Term Research Project SYRF 22703

[†]Verimag is a joint laboratory of Université Joseph Fourier (Grenoble I), CNRS and INPG

guages, like Esterel [?], Lustre [?], the Statecharts [Har87, Har88] or Argos [Mar92], whose semantics rely more or less on the same model. This model is the notion of automaton, or labeled transition system, which is very well adapted to the representation of reactive behaviours.

The paper is structured as follows: Section **??** is a general introduction to reactive systems and the synchronous approach ; Section **??** presents Argos (a purely synchronous and compositional version of Statecharts) ; Section **??** presents the language of Mode-Automata (the result of mixing Argos with dataflow equations, for the description of running modes in reactive systems).

2 Reactive Systems and the Synchronous Approach

2.1 Examples of Reactive Systems

Typical examples of reactive systems include the *regulation* systems, and the *event-driven* systems.

2.2 Typical synchronous program for a reactive system

Synchronous languages are high level languages that should be compiled into sequential code of the following form:

```
Initializations
while true
    Get input $i$
    $\boxed{\texttt{Compute output } o}$
    Emit output $o$
```

A pass in the loop corresponds to an instant of the program discrete time. At instant n, the program reads input i_n, and outputs the output o_n.

In general, the relation between inputs and outputs is as follows: $\forall n,\ o_n = f(i_0, i_1, ..., i_n)$, which raises several remarks:

First, o_n may not depend on future inputs $i_{n+...}$; this is called the *causality* constraint.

Second, o_n may depend on i_n istself, not only on *previous* inputs; this is called the *synchrony hypothesis.*

Finally, if no restriction on f is made, the history of inputs is unbounded, and the above function cannot be programmed with a statically allocated memory.

For reactive systems and embedded systems, we definitely do not want the programs to have dynamic errors like "no more memory, allocation failed", and we require that the memory needed for f be bounded.

This means that the output at each instant n does not depend on the whole history of inputs, but only on a bounded abstraction of it: $\forall n,\ o_n = f(\text{Abs}(i_0, i_1, ..., i_n$

where Abs is the *abstraction function*, with the following property: $\exists B \ \mid Image(\text{Abs}) \mid < B$

If the function Abs induces an equivalence of input histories that is a congruence for the rightmost extension of such sequences:

$$\exists g,\ Abs(i_0, i_1, ..., i_n) = g(Abs(i_0, i_1, ..., i_{n-1}), i_n)$$

then the typical program becomes:

```
Initialization of the memory $M$
while true
    *Invariant :*      $M = Abs(i_0, i_1, ..., i_{n-1})$
    Get input          $i_n$
    Compute output     $o_n = f(M, i_n)$
    Update memory      $M = g(M, i_n)$
    *Property here :*  $M = Abs(i_0, i_1, ..., i_n)$
    Emit output        $o_n$
```

2.3 Synchronous languages

All the synchronous languages are devoted to the description of such programs or, better, to the description of their so-called *reactive kernel*, i.e. the f and g functions of the above algorithm. Following these ideas, they are mainly two approaches that can be adopted in order to design a high level language for reactive systems.

The state-transition paradigm can be considered to be too low level. It is used only at the model level, and the language constructs are chosen according to another way of designing the reactive systems. This is the case for Esterel, Lustre and Signal. The designer does not "think" in terms of automata. Lustre and Signal are dataflow languages, and describe such programs by sets of equations relating the flows of values of M, i and o along time. Esterel is a textual imperative language in which specific construct correspond to the access to inputs and emission of outputs, and other constructs to control structures like watchdogs, etc.

The other approach is to consider that the state-transition paradigm is powerful enough, in the case of reactive systems, to be the basis of a high level language. This means that the user will have to "think" in terms of automata. In Statecharts and Argos, the state-transition paradigm is considered to be powerful enough to be the basis of a high level language. Statecharts and Argos deal with explicit memory: since M is bounded, each value may be represented by a *state*. In this case, the f and g functions above correspond to the output and transition functions of a Mealy machine.

The limits of this approach are reached quickly: the design of a complex reactive system, as a single automaton, is completely unrealistic. A way of *structuring* automata descriptions has to be found.

3 Argos

A system like the digital watch is described in a very convenient manner by seing the running modes of the watch as *states*, and the changing of modes as *transitions*. States can be `Watch`, `Stopwatch`, `Setwatch` and `Alarm`, and the user may change modes by depressing buttons *Upper Left* or *Lower Left*. This constitutes the *physical* events. We associate to them the *logical* events `UL` and `LL`, to be used in the description of the watch, as the labeling of transitions. The description of a reactive system is the description of its reactive *kernel*, which deals with logical events. The interface between the environment and the reactive kernel, which translates physical input events into logical ones — and logical output events into physical output events, has to be described in another language.

The Argos syntax is originally inspired from that of Statecharts, but the language constructs are different. To the author's opinion, Statecharts make a more extensive use of the graphical syntax than Argos. For small systems, the Statecharts representation may be more concise than the Argos one, and, in this sense, more readable.

But concision is not the only criterion for readability, especially when the size of the systems we consider grows. Even if a picture is worth a thousand words, there will always be systems big and complex enough that cannot be represented on one physical paper sheet. So the language must provide a way to cut the representation into smaller parts. The point is that this operation must not be only syntactical. Cutting a representation *a posteriori*, without taking into account the semantics of the system described will make it unreadable (the same is true for textual languages: listings are cut between procedures, or between control structures, if procedures are too long). The representation must have a graphical structure, in order to be cuttable, and this structure must be the semantical structure of the system, for the different parts to be meaningful.

In the Statecharts, this is clearly not possible, because there is no way the representation of a complex system can be cut into small graphical parts which represent sub-systems (see section 3.4, which gives some hints to understand this). A system has to be designed globally, and represented globally.

Argos proposes a different use of the graphical representation of automata. A program is either an automaton, or the result of applying a unary, binary or n-ary operator to program operands. Each automaton is drawn with boxes and arrows. Each operator is given a graphical syntax and the representation of operators constitutes the squeletton of the program: its graphical and semantical structure.

3.1 From Statecharts to Argos

The motivation for Argos was to use a set of constructs taken from Statecharts, in order to describe Mealy machines. It is really important to remark that the target of the semantics was chosen before the language was designed.

A lot of semantics for Statecharts have been proposed. Often they are described as functions from statecharts to a domain especially designed for that.

3.1.1 Designing a concrete grammar of Statecharts

The first thing to be done is to design a grammar of Statecharts. A lot of papers on Statecharts use a complex *global* description of Statecharts, and there is no hope to design a syntax-directed semantics from such a description. Some other papers propose to describe complex Statecharts — including inter-level transitions — as the combinations of a small set of basic graphical objects. One of these basic objects is a state with dangling input and output arrows. One may obtain a concrete grammar of the graphical objects that constitute Statecharts in such a way, but the problem will be that there is no chance to associate a meaning to such basic objects. What could the reactive behaviour of a dangling arrow be?

It appeared that a necessary condition for being able to define a grammar of Statecharts, with meaningful basic objects, was to get rid of inter-level transitions. Without interlevel transitions, a Statecharts may be viewed as a set of automata (states and transitions can be properly grouped at one level), involved in two kinds of compositions: the *parallel composition*, depicted by a dashed line between two (possibly composed) objects; the *hierarchic composition*, depicted by drawing a (possibly composed) object *into* the state of another one.

Removing interlevel arrows is sufficient for obtaining a concrete grammar of Statecharts. We will see below that this is not sufficient for obtaining a compositional semantics, but it is still necessary for that.

other features we do not deal with: history input, conditional arrows, ...

3.1.2 Requiring a Compositional Semantics

The story of the 42 semantics of Statecharts variant taught us that every semantics may be made compositional, if one does not care about the domain of the semantic function.

3.2 A set of operations on Boolean Mealy machines

The parallel composition, which has to be associative and commutative.

The priority between the transitions sourced in the same state: some graphical interfaces for automaton-based languages use a graphical version of the `case` construct of sequential languages: the order of the transitions is relevant, starting from "noon", and turning as a watch ...

Hierarchy: a good place for encapsulating subprograms like procedures of sequential languages. Conversely, an automaton should not be split into several parts, described on distinct pages of the program.

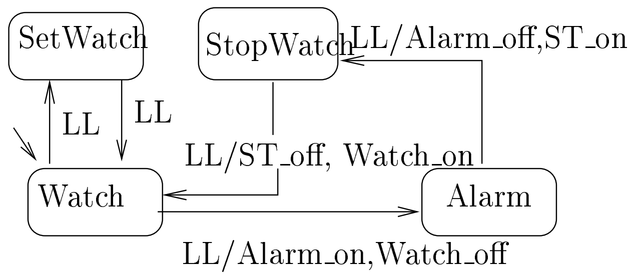


Figure 1: A reactive behaviour

3.3 Argos

3.3.1 Basic behaviours

A simple reactive behaviour may be described by a *labeled transition system* as shown by figure 1. The transition system has one *initial* state. Transition labels are made of two parts: the *input* part I, and the *output* part O. The complete label is denoted by I/O. Both parts are built upon a set `E` of elementary interactions with the environment, called *events.* The input part is a conjunction of events or negations of events. It describe a condition to be fullfilled by the environment in order to make the system react. The output part gives the events the system outputs to its environment, when reacting to a given input. In the sequel, we shall refer to the buttons of the watch as `LL, LR, UL` and `UR`, standing for: *Lower Left*, *Lower Right*, *Upper Left* and *Upper Right.* We introduce logical output events to control the physical state of some lamps, used to show the current running mode of the watch (in a more realistic description of the watch, the changing of modes makes the display of the watch change, but it is not reduced to "lamps"; see below). The `X` lamp may be switched on and off with events `X_on` and `X_off` respectively. The `Watch` lamp is initially on.

The first advantage of the graphical representation is that it makes the detection of non-determinism very easy; it is an important notion for reactive systems. In spite of the inherent non-determinism in the description of the environment, the programs should describe deterministic behaviours. In this framework, non-determinism of a reactive behaviour is simply the existence of two transitions sourced in the same state, with the same input part, and different output parts and/or target states. In the textual representation of an automaton, the labels of the transitions sourced in a given state are not necessarily grouped, and conflicts like non-determinism may be difficult to read. In the example of figure 1, the reaction of the watch to button `LL` when in `Watch` mode is not deterministic. A single button cannot be used for two functions, and one `LL` input of the two transitions sourced in `Watch` mode should be replaced by a new logical event, related to a new button (see figure 2).

3.3.2 The Argos constructs

This section constitutes a very brief presentation of the language constructs, focusing on the relation between semantical and graphical constraints. It is necessarily incomplete concerning the precise semantical definition of the language. However, the following points can give sufficient hints for the understanding of the paper. The graphical representation is only a *syntax*, with meaningful aspects (i.e. connections between boxes and arrows) and meaningless ones (i.e. size of boxes). There exists an equivalent textual syntax. There exists a unique underlying model (the compiled form of a program) for each valid graphical representation. There exists a graphical representation for each valid model, since this is an automaton, and all automata can be represented in the Argos syntax. There exists several graphical representations for any valid model; they may differ on meaningless aspects (as two identical textual programs may have texts which differ on the number of blank lines), or on meaningful ones (the compiled form of a program may be obtained by using different constructs at the language level).

Parallel composition and local events

We consider now a more realistic digital watch. Observe figure 2. We describe the watch as a set of *parallel* components, which communicate with each other. The *interface* we described partially is completed, and we add special components in order to control the display of the watch. Components are separated by dashed lines. In this example, they are all automata, but they could be composed systems as well. The parallel *operator* is defined formally as a binary operator, but it is commutative and associative, so the graphical syntax illustrated by the figure makes sense: there is no need for explicit parenthesis when there are more than two components. Associativity and commutativity of the parallel composition is a rather strong semantical constraint, but explicit parenthesis (for instance with rectangles to group components together) would make the graphical syntax too complex.

The interface deals with input events (the four buttons) and contains exactly the information which is necessary to interpret them correctly. The interface is the only component of the system which "*knows*" that `LL` when in `Watch` mode means "*change mode*", `UL` when in `Watch` or `Alarm` mode means "*enter the corresponding set mode*", `UR` means "*toggle alarm indicator*", but only when in `Alarm` mode, etc. Other components do not deal with input events directly. The interface performs a translation from the logical events which correspond to buttons, to events like `comAL` (for "*commute alarm indicator*") or `comMODE` (for "*commute mode*"). Conversely, the interface does not deal with the display of the watch. Other components do.

The *Alarm indicator* component is used to memorize the state of the alarm indicator. When changing states on `comAL`, it switches on or off a little lamp of the watch display, which can, for instance, display a little bell when alarm is on. This lamp is initially off. The *Main display* component maintains the state of the main numeric display. It can show either Hours, Minutes and Seconds, or Hours and Minutes only, or Minutes, Seconds and milliseconds.

Communication between components is done by the events which are output by one component and input by another one, like `comAL`. The semantics of the communication mechanism is defined formally as a *synchronous broadcast* [?]. When a component outputs an event, it broadcasts it towards its whole environment (the other components, and the global environment of the system). Similarly, its inputs may be inputs from the global environment, or events output by another component.

The description of a reactive system often introduces a lot of events which are used only for internal communication. Argos provides the designer with a way to declare *local events* (see figure 2). The rectangular box labeled with `comAL, comMODE` defines the scope of events `comAL, comMODE`: inside the box, these events can be used as inputs or outputs between components. But, when used as input, they cannot come from the environment outside the box, and when output, they are not visible outside the box.

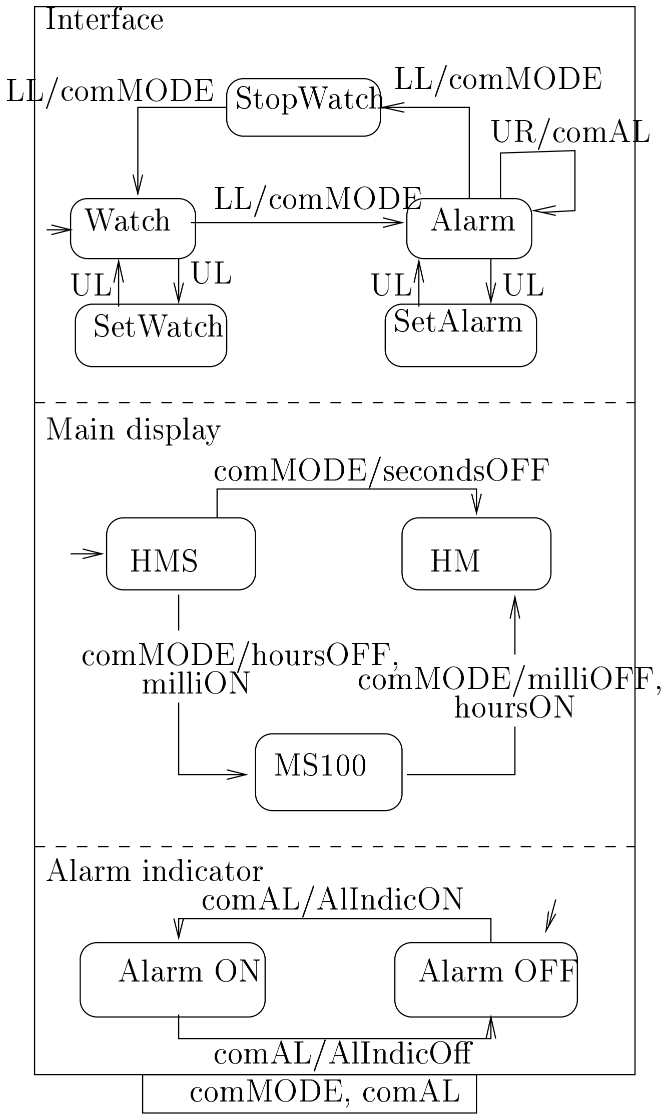


Figure 2: interface and ressources of the watch,
an example of parallel components

Refinement operation

We focus on the interface component now. As described above, it does not express all the meaning of the buttons. Figure 3 shows a complete description of the interface. We introduce the following events: `comLAP` and `comRUN` to control the stopwatch behaviour, `comCH` to commute chime mode (when chime is active, the watch beeps each hour), `incr`x`_H` and `incr`x`_A` to control the time keeper component.

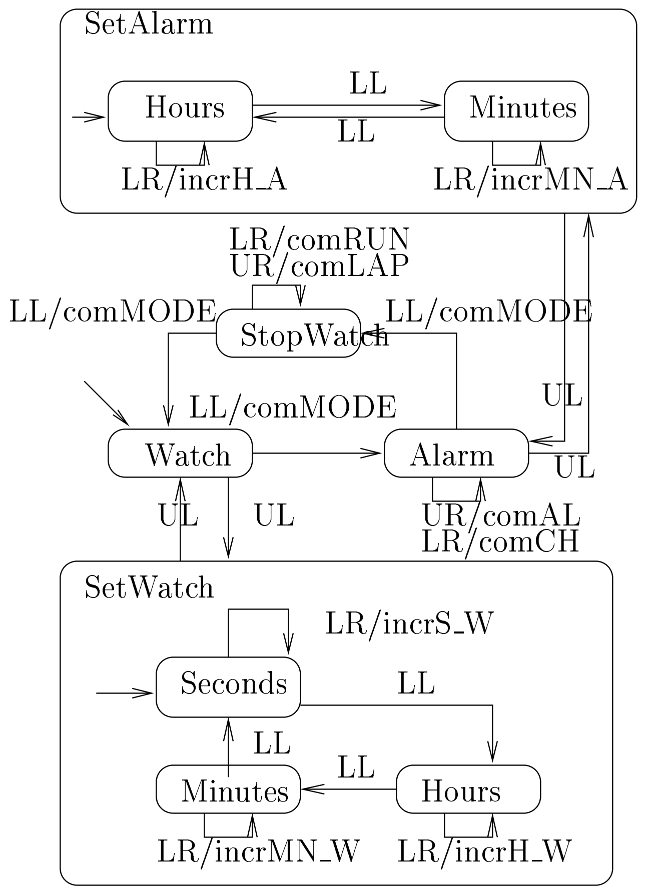


Figure 3: interface of the watch,
an example of refinement

The `SetWatch` and the `SetAlarm` states are *refined* by sub-systems. Observe the `SetWatch` state. When the watch is in `Watch` mode, and the users presses `UL`, the `SetWatch` state is entered, and the sub-system which refines it is *started*, in its initial state, i.e. `Seconds`. Then, two sub-systems are active:

- the *controller* with states `Watch, Alarm, SetWatch, SetAlarm` and `StopWatch`, which can react to `UL`
- the *controlled* sub-system with states `Seconds, Hours` and `Minutes` which can react to `LL` or `LR`. `LL` is used to change fields, and `LR` is used to increment a field. The incrementation outputs appropriate events to a time keeper component not represented here.

When the controller reacts to `UL`, the controlled sub-system is *killed*, and the watch reenters `Watch` mode.

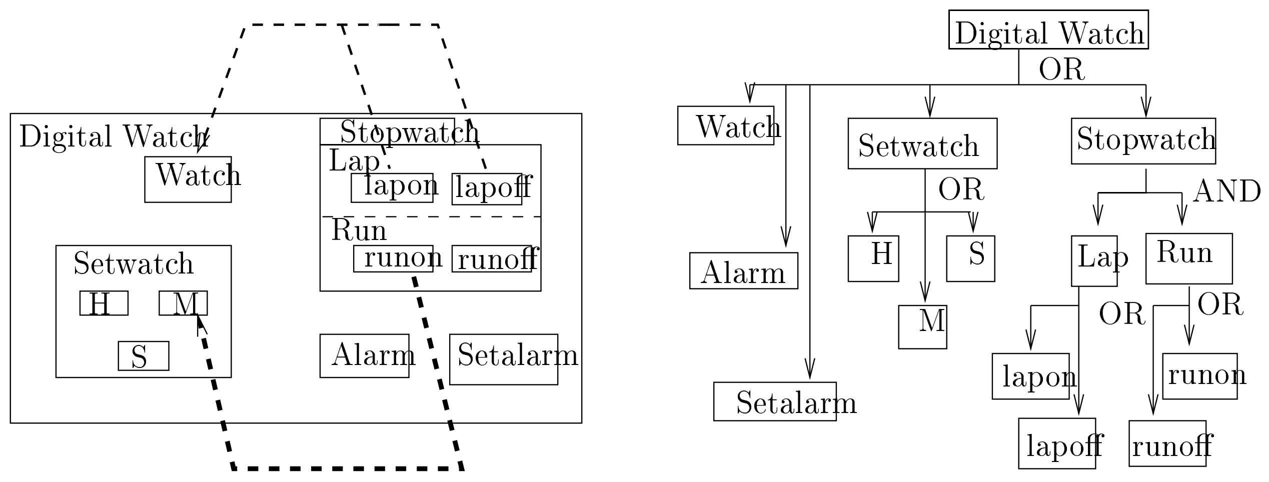


Figure 4: a Statechart and the corresponding state space decomposition

3.4 Comparison with Statecharts

Due to its graphical syntax, Argos may seem to be very similar to Statecharts [Har87], one of the possible applications of the notion of Higraph presented in [Har88]. This paper gives a lot of ideas about the constructs that can be introduced in a graphical syntax in order to improve the representation of a big and complex state-transition diagram. It introduces the notions of orthogonality (cartesian product) and hierarchy (ability to cluster states into a macro-state).

When applied to the description of reactive systems, these ideas give Statecharts. They lead quite naturally to design methods and programming style which are very different from that of Argos. Indeed, Statecharts are very well adapted to the decomposition of the state space of the system, as an AND/OR tree. AND nodes correspond to orthogonality, and OR nodes to hierarchy. The intuitive meaning is the following: when the system is in an OR state, it is in one of it sub-states; when it is in an AND state, it is in all its sub-states at the same time. Figure 4 illustrates the state space structure for a version of the watch where the behaviour of the `StopWatch` state is detailed. Two examples of authorized arrows are given in the Statechart. The *configurations* (global states) of the watch are: `Watch`, `H`, `M`, `S`, `Alarm`, `SetAlarm`, `lapon` $\times$ `runon`, `lapon` $\times$ `runoff`, `lapoff` $\times$ `runon` and `lapoff` $\times$ `runoff`.

The behaviour of the watch, which is a set of transitions between configurations, is expressed by drawing arrows between boxes of the state space representation. Statecharts allow arrows between any two boxes of the representation, even at different levels. The upper dashed arrow of figure 4 is a transition between configurations `lapoff` $\times$ `runoff` and `Watch`. The lower dashed arrow is a a *set* of transitions: `lapon` $\times$ `runon` to `M`, and `lapoff` $\times$ `runon` to `M`.

This is the feature which makes the Statecharts more concise than Argos sometimes. But this is also the reason why there is no way to distinguish sub-systems in a big Statechart, and, consequently, it is very difficult to cut the figure into meaningful parts. In some sense, *inter-level* transitions can be compared to GOTOs in classical sequential languages.

3.5 Recovering some Statecharts Features in Argos

From this set of constructs, using the properties of perfect synchrony, one may design macro-notations for some features of Statecharts we had to reject in the first step. Inter-level transitions are of this kind.

3.6 Related work

SyncCharts, Argos+Esterel

3.7 Implementation concerns

3.7.1 Editing Argos programs

Le résumé des reflexions et commentaires tirés de ces expériences tient en quelques points importants. Tout d'abord, pour réaliser un éditeur de langage graphique, il a deux solution extrêmes, entre lesquelles on peut imaginer toute une variété de solutions mixtes :

Analyse syntaxique 2D : on peut utiliser un éditeur graphique général, et réaliser une analyse de la figure pour y repérer les objets du langage (c'est exactement l'approche suivie pour les langages séquentiels, où l'on invente une syntaxe concrète à base de séparateurs et de mots-clés pour représenter linéairement, sans perte d'information, une structure typiquement arborescente. L'analyse syntaxique consiste alors à reconstruire l'arbre abstrait à partir du texte linéaire). Il est préférable d'utiliser un éditeur qui fournit des objets de base comme les rectangles et les lignes, plutôt qu'un éditeur de bitmap. Si tous les éléments de syntaxe concrète du langage correspondent à des objets graphiques connus de l'éditeur, une partie du travail de reconnaissance est épargné. Reste toutefois à repérer l'organisation de ces objets (typiquement, dans un éditeur d'automate, quel critère de proximité peut-on utiliser pour reconnaître l'association flèche-texte qui définit une transition ?). Cette approche, qui peut sembler sans espoir, a pourtant été utilisée dans d'autres contextes, en particulier avec l'éditeur Idraw de l'INRIA qui produit du postscript commenté par les types d'objets. D'autre part il existe des travaux théoriques sur la généralisation des algorithmes d'analyse syntaxique de texte à des sources 2D. Voir par exemple [CTC91, CTC90], dans lesquels on utilise une relation spatiale entre objets.

Edition syntaxique : on peut développer un éditeur syntaxique, basé sur un outil de dessin. On se trouve alors confronté à trois types de problèmes :

- des problèmes de dessin plus ou moins interactif : typiquement, dans un éditeur d'automates, il est bien agréable de disposer d'un placeur automatique de transitions, et on aimerait que le déplacement de la boîte qui représente un état s'accompagne d'un déplacement harmonieux des flèches représentant les transitions qui y sont attachées
- des problèmes classiques de manipulation d'arbre abstrait : doit-on imposer une construction ascendante ou descendante, quelles opérations de suppression de sous-arbre peut-on autoriser...? sont quelques unes des questions qui se posent au concepteur.
- des problèmes de choix du degré d'analyse garanti par l'éditeur : peut-on se contenter d'un éditeur qui ne gère que les aspects de syntaxe hors-contexte, ou doit-on également garantir une analyse des aspects de syntaxe contextuelle (ou sémantique statique) ? Dans ce dernier cas, à quels blocs s'applique l'analyse, et quand ? (La réponse "*partout, tout le temps*" conduit à définir un éditeur dans lequel tout objet à peine ébauché doit être correctement constitué avant de pouvoir être utilisé comme composant d'un objet plus complexe. C'est un comportement extrêmement contraignant, pour ne pas dire pénible).

Nous n'avons jusque là considéré que les aspects de création et modification de programmes. Un autre aspect important concerne les choix de présentation des programmes. En effet, comme exposé dans [Mar91] pour ARGOS, un langage graphique est intrinsèquement peu concis[1]. Il faut donc prévoir des techniques de présentation partielle des programmes — par exemple en ne montrant pas systématiquement le contenu des états raffinés — et imaginer des opérations de manipulation de la présentation — zoom sur un composant raffinant, etc.

Related work: Autograph, Arged (GMD, Sankt Augustin, projet Synchronie), Editeur des SyncCharts (Université de Nice)

3.7.2 Compiling Argos programs

4 Mode-automata = Argos + dataflow equations

4.1 Motivations

In the field of reactive system programming, engineers who have to design control laws and their discrete form were used to block-diagrams. dataflow synchronous languages like Lustre [BCH+85, CHPP87] or Signal [GBBG85] offer a syntax similar to block-diagrams, and can be efficiently compiled into C code, for instance. Designing

[1]Contrairement à l'idée reçue qui veut qu'*un dessin vaille mieux qu'un long discours*

a system that clearly exhibits several “independent” *running modes* is not difficult since the mode structure can be encoded explicitly with the available dataflow constructs. However the mode structure is no longer readable in the resulting program; modifying it is error prone, and it cannot be used to improve the quality of the generated code.

We proposed to introduce a special construct devoted to the expression of a mode structure in a reactive system [MR97, MR98]. We called it *mode-automaton*, for it is basically an automaton whose states are labeled by dataflow programs. We also proposed a set of operations that allow the composition of several mode-automata (parallel and hierarchic compositions taken from Argos [Mar92]), and we studied the properties of our model, like the existence of a congruence of mode-automata for instance, as well as implementation issues.

4.2 Overview of Lustre

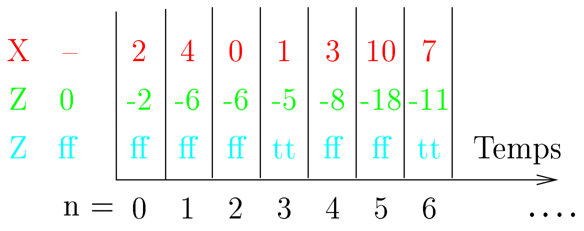
Here is a very simple example of a Lustre program.

```
INPUT   :  X int
OUTPUT  :  Y int init 0
LOCAL   :  Z bool init false
Equations
Y = pre( Y) + (if  Z then X else − X)
Z = if  X modulo 2 = 0 then pre(Z) else not pre( Z)
```

The input, output and local *variables* are names for *flows*. A flow is an infinite sequence of values, indexed by integers. The index is also called the *instant*: in a Lustre program we have a discrete notion of time.

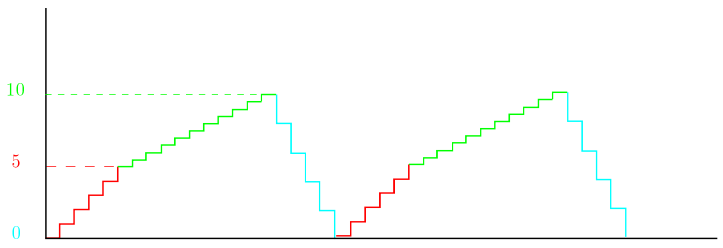
The behaviour of the program may be observed on the flowing picture:

Equations are valid at each *instant*. `V = exp` *means* $\forall n \geq 0 . V_n = exp_n$



4.3 A Construct for Dealing with Modes

Suppose we have to describe, in Lustre, the behaviour of a flow X, as pictured below:



This system clearly has three distinct evolution patterns: at the beginning, X increases by one at each instant; then it increases by a half; then it decreases by two.

The behaviour of the variable X is very easy to describe in Lustre, using the conditional structure. But we would like to provide the users with a specific construct, that allow to deal with modes explicitly.

The idea is that the three distinct evolution patterns should be described independently from each other :

4.4 A Very Simple Mode-Automaton

The mode-automaton of figure 2 describes a program that outputs an integer X. The initial value is 0. Then, the program has two *modes*: an incrementing mode, and a decrementing one. Changing modes is done according to the value reached by variable X: when it reaches 10, the mode is switched to "decrementing"; when X reaches 0 again, the mode is switched to "incrementing".

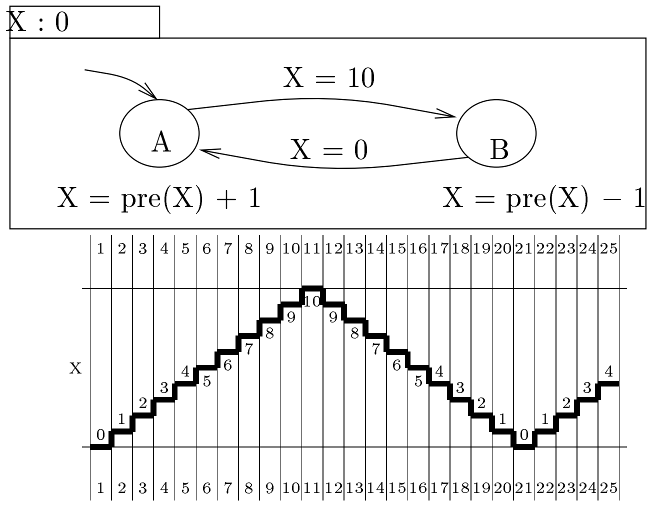


Figure 2: Mode-automata, a simple example: the program and its temporal behaviour

4.5 Composing Mode-Automata

4.6 A Realistic Example

4.7 Implementation Concerns

4.7.1 Editing Mode-Automata

parler de Lustre/SCADE - already a graphical and syntactic editor for dataflow programs, with a synchronization between the graphical and the Lustre textual forms of a program.

4.7.2 Compiling Mode-Automata

4.7.3 Simulating Mode-Automata: the tool Sim2chro

5 Conclusions and General Comments

References

[BCH+85] J-L. Bergerand, P. Caspi, N. Halbwachs, D. Pilaud, and E. Pilaud. Outline of a real time data-flow language. In *Real Time Systems Symposium*, San Diego, September 1985.

[BG88] G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. Technical report, INRIA report 842, 1988.

[CHPP87] P. Caspi, N. Halbwachs, D. Pilaud, and J. Plaice. LUSTRE, a declarative language for programming synchronous systems. In *14th Symposium on Principles of Programming Languages*, Munich, January 1987.

[CTC90] G. Costagliola, M. Tomita, and S. Chang. Dr parsers: a generalization of lr parsers. In *IEEE Workshop on Visual Languages*, pages 174–180, Skokie (Illinois), October 1990. IEEE Computer Society Press.

[CTC91] G. Costagliola, M. Tomita, and S. Chang. A generalized parser for 2-d languages. In *IEEE Workshop on Visual Languages*, pages 98–104, Kobe (Japan), October 1991. IEEE Computer Society Press.

[GBBG85] P. Le Guernic, A. Benveniste, P. Bournai, and T. Gauthier. Signal: A data flow oriented language for signal processing. Technical report, IRISA report 246, IRISA, Rennes, France, 1985.

[Har87] D. Harel. Statecharts : A visual approach to complex systems. *Science of Computer Programming*, 8:231–275, 1987.

[Har88] D. Harel. On visual formalisms. *CACM*, 31, 1988.

[HP88] D. Harel and A. Pnueli. On the development of reactive systems. In *Logic and Models of Concurrent Systems, NATO Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems*, volume 13. NATO ASI series F, Springer Verlag, 1988.

[Mar91] F. Maraninchi. The argos language: Graphical representation of automata and description of reactive systems. In *IEEE Workshop on Visual Languages*, Kobe, Japan, October 1991.

[Mar92] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *CONCUR*. LNCS 630, Springer Verlag, August 1992.

[MR97] F. Maraninchi and Y. Rémond. Compositionality criteria for defining mixed-styles synchronous languages (invited paper). In *International Symposium: Compositionality - The Significant Difference*, Malente (Holstein, Germany), September 1997. Springer Verlag.

[MR98] F. Maraninchi and Y. Rémond. Mode-automata: About modes and states for reactive systems. In *European Symposium On Programming*, Lisbon (Portugal), March 1998. Springer Verlag, LNCS 1381.

Visualising business processes

Paul Oude Luttighuis

Marc Lankhorst

Rob van de Wetering

René Bal

Harmen van den Berg

Contact:

Paul Oude Luttighuis

Telematics Institute

P.O. Box 589, 7500 AN Enschede, The Netherlands

Phone +31 53 485 0417, Fax +31 53 485 04 00

e-mail: `luttighu@telin.nl`

Abstract

Graphical representations of business process models are an important means by which business architects can grasp the inherent complexity of business processes. This paper reports on the visualisation features of AMBER, the graphical and formal business process language of the Testbed project, and Testbed Studio, the toolkit developed by the project. The visualisation features fulfil explicit requirements for visualisation of business process models, which are based on an investigation of a business architect's main activities. The paper presents the three main areas in which visualisation is used in AMBER and Testbed Studio: (1) the graphical representation of the language itself, (2) the representation conventions suggested in the Testbed handbook, and (3) a set of business process views in Testbed Studio.

1. Introduction

Organisations are complex for those who try to master them. They involve different customer groups, business units, people, resources, and systems. They stretch over numerous different processes that interact in a seemingly chaotic way. When trying to change business processes within organisations, one is confronted with that inherent complexity [1].

The Testbed project develops methods, techniques, and tools to handle change of business processes, particularly in the financial services sector [3]. A main objective is to give *insight* in the structure of business processes and their relations. This insight can be obtained by creating *business process models* that clearly and precisely represent the *essence* of the business organisation. These models should encompass different levels of organisational detail, thus allowing to find bottlenecks and to assess the consequences of proposed changes for the customers and the organisation itself. Formal methods allow for detailed analysis of models and facilitate tool support.

The results of the Testbed project include a graphical business process modelling language, called AMBER [2], and a tool environment, called Testbed Studio, with which AMBER models can be created, manipulated, analysed, and viewed from different perspectives.

One set of considerations in designing AMBER concerned its graphical representation of business process models and their analysis results. Adequate visualisation of these was considered to be an important contribution to Testbed's objective: providing means for tackling complexity in business processes.

In this paper, we focus our attention on these visualisation aspects. Section 2 starts with an investigation of the role of graphical representations in business process modelling. This leads to the identification of visualisation requirements for AMBER and Testbed Studio, based on a model of business process change. Section 3 then presents AMBER, with particular attention to the graphical representation. Two other graphical features are treated in section 4 —namely representation conventions— and section 5 —namely views, which show business models from different perspectives. In section 6, we show how the visualisation features of AMBER match the requirements presented. We conclude with a summarising discussion and directions for further work in section 7.

It should be noted that the focus of this paper is on the visualisation of business process models, not on the entire graphical user interface (GUI) of Testbed Studio. Still, some GUI design principles [5] apply to visualisation of business processes as well.

2. Requirements for visualising business processes

2.1 The role of visualisation in AMBER

As mentioned, the semantics of AMBER have been formally specified. The main objective of this formal semantics is to enable automated analysis of business processes by mapping the language concepts onto a mathematical domain. In fact, for different types of analysis, different formal semantics may be provided. The role of a graphical representation of AMBER however is not to visualise formal aspects of business processes. This is because the user of AMBER —the business architect— should be shielded from the formal details of the language, be they partial-order models or Petri-nets. The representation of AMBER should convey to the business architect the meaning of the language elements. The graphical representation should allow easy interpretation of the models by the business architect, such that the interpretation matches the intended meaning. This is depicted in Figure 1.

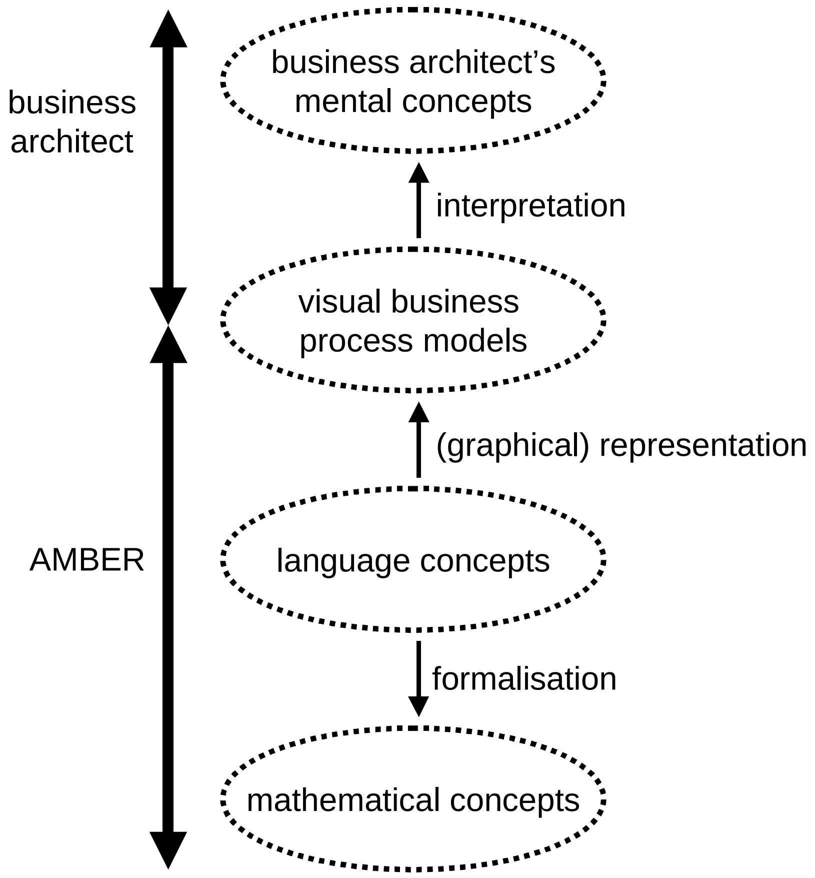

business architect
business architect's mental concepts
interpretation
visual business process models
(graphical) representation
AMBER
language concepts
formalisation
mathematical concepts

Figure 1 — The role of graphical representation.

It is beyond debate that this representation be a graphical one. As opposed to one-dimensional textual representations, two-dimensional graphical representations are richer in structuring possibilities. In fact, business processes involve even more than two dimensions, which are interdependent in complex ways. For instance, there are many actors, activities and data involved in business processes. The same data may be manipulated by many different activities, which in their turn may manipulate other data as well. Actors may carry out different activities and activities may be carried out by different actors. Activities also have mutual dependencies. They may access the same data at the same time or they may be ordered in complex ways.

Textual representations must make a mapping of this complexity into a single dimension and therefore require typical elements such as delimiters, operators and identifiers. In formal terms, no meaning is lost, but the structure of a large business process is difficult to retrieve from a textual representation. In some cases layout may help, for instance by using indentation to express nested structures.

Take for instance the parallelism occurring in business processes. This parallelism is naturally expressed in graphical languages by paralleling the representations of the constituents, whereas a textual representation must resort to an explicit operator, such as ||. In graphical languages, associations between model elements can be indicated naturally by using an arrow or line between them, instead of symbolic referencing, which should be used in textual representations. In addition, graphical symbols have a visual appearance, as well as a name. The existence of the symbol in the model declares the object, its visual appearance classifies it, whereas its name identifies it. This is close to the way humans perceive real-world objects. Textual representations have to use separate mechanisms: declarations, explicit typing, and identifiers. Only where textual representation is usual (as in names) or graphical representation would be too verbose (as in simple formulas), AMBER chooses a textual representation.

In this sense, textual representations are more abstract than graphical ones. However, the language concepts in AMBER are abstract concepts, not concrete ones, because only abstraction helps to master complexity [1,8]. This is why the graphical representation of AMBER uses abstract symbols, such as circles, arrows, and so on, instead of detailed icons. In this sense, the semantic conciseness of the concepts should be reflected in the simplicity of the symbol. The use of icons implies a risk of over-interpretation, because they may contain irrelevant details and hence unintentionally suggest additional meaning.

With the business architect being the main user of AMBER, we first investigate his main activities in order to formulate visualisation requirements for AMBER.

2.2 Activities of a business architect

In this section, we will take a closer look at these activities and couple them with specific visualisation requirements. In following sections, we will then show how different visualisation features of Testbed Studio comply with these requirements.

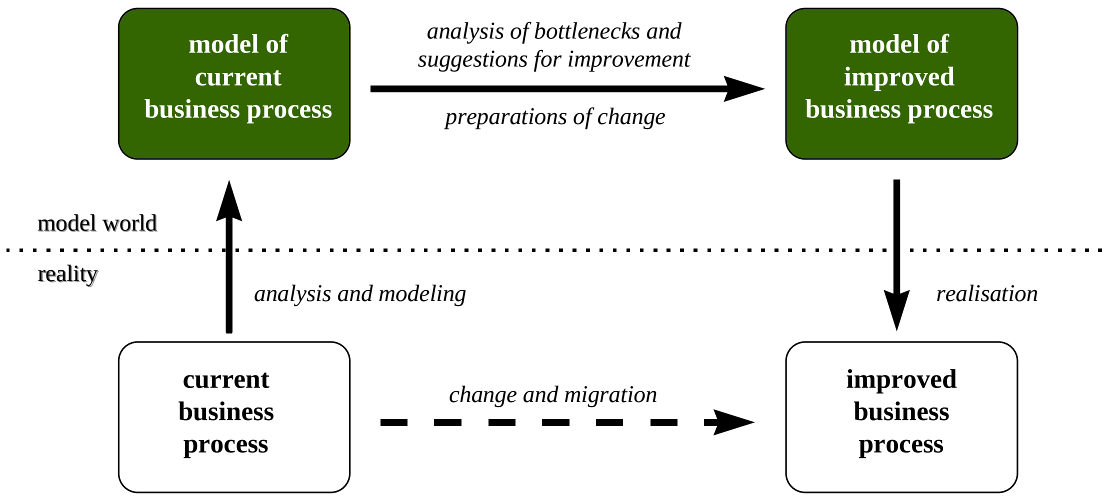

model of current business process
analysis of bottlenecks and suggestions for improvement
preparations of change
model of improved business process
model world
reality
analysis and modeling
realisation
current business process
change and migration
improved business process

Figure 2 — Change of business processes supported by Testbed.

The current business organisation is the starting point for a change in business processes. In order to match it to the company mission and change objective, it is important to gain insight into the current situation and use it as a baseline. An analysis of the current business organisation provides this insight. Figure 2 depicts the over-all Testbed approach. Several alternative new situations can be designed and repeatedly analysis and compared. Off-line analysis cuts down on the costs and risks of business process change. Stepwise migration is supported by a roadmap of consecutive models of intermediate situations. These models can be validated and used for all sorts of computations to analyse, visualise and justify changes.

Concluding, we identify the following main activities of the business architect:

– modelling current business processes as well as future alternatives,
– analysing business process models,
– planning and guiding business process migration, and
– communication with different stakeholders in any of the above activities.

2.3 Visualisation requirements

Modelling

First, when *modelling* the current business process, a business architect should be able to tackle the inherent complexity of business processes [1] and get a grip on their size, their many facets and their variety. For this purpose, the business architect is offered a *graphical language for business process modelling*. Appropriate visualisation should help the business architect to create clear models. We will introduce the language in section 3.

In our visualisation choices for AMBER, we were led by considerations of unambiguity, organisation, economy, orthogonality, and external consistency (see also [5]). In general, it is impossible to entirely adhere to all these guidelines simultaneously. Especially external consistency may be difficult to accomplish simultaneously with economy and orthogonality.

Unambiguity

There should be a single (formal) meaning for every symbol and model. Without this requirement, the main role of graphical representation —conveying a model's single meaning— is jeopardised.

Organisation

The business architect should be offered means to organise business process models, not only conceptually but also graphically. No matter how sophisticated the language concepts, no complex business process can be grasped in a simple model. Visually arranging the model's representation further aids in mastering complexity.

Economy

Business process models should be conveyed by modest means, simple symbols, and few symbols. Many symbols or complex symbols (such as icons) increase visual complexity and hence decrease the accessibility of a representation.

Orthogonality

Orthogonality (also called internal consistency [5]) requires that differences and similarities between language concepts should be reflected in their graphical representations. For instance, consider a language with two distinct sets of related concepts. The representation of the concepts within a single set should share a visual characteristic, such as their colour or an aspect of their form. This allows the business architect to associate meaning with individual visual characteristics of a symbol or representation. He can then correctly reconstruct the meaning of a representation by individually interpreting its visual characteristics and combining these interpretations.

External consistency

A business architect’s expectations and experience should be adhered to. This lowers the business architect’s threshold for learning the language and limits the size and complexity of his entire set of mental concepts.

General visualisation choices

Formal and informal meaning

Next to the strict meaning of a business model, visualisation features may help to convey informal meaning to or between business architects and other stakeholders. Our decision has been to have shape carry meaning. That is, for each concept in the language, as well as for each way of associating them, there should be a uniquely shaped symbol.

Colour therefore does not carry strict meaning: a red circle has the same meaning as a blue one. Still, colour is a powerful visualisation means and is used extensively in our views. Furthermore, the editing tool of Testbed uses different default colours for different concepts. This may be seen as a form of redundant visualisation, which is often recommended as a means to clarify the meaning of models. However, the business architect may alter the colour of any symbol according to his wishes.

Size and orientation

The (absolute or relative) size of symbols nor their (relative) orientation may carry any formal meaning, since we want models to be rotation and magnification invariant. The shapes of the symbols are simple. Although this might make them more abstract than, say, icons, it helps to keep models light and less confusing, as well as easy to draw by hand

Associations

In general, associations between concepts can be graphically represented in a number of ways:

- by having one symbol include the other,
- by having the symbols overlap or touch,
- by connecting them with another symbol, or
- by placing the symbols in each other's proximity.

Inclusion can be used for hierarchical relations between concepts, as it is an asymmetric association. Touching, overlap or proximity can be used for non-hierarchical associations, either symmetric or where the difference between the model elements is clear from the symbols themselves.

Proximity has its drawbacks, as it is not precise. Especially in dense models, with many symbols, it may be difficult to decide (at least at first glance) which symbols are associated, or proximity among symbols may suggest relations which are not there. On the other hand, in case of associations between textual language elements on the one hand and one-dimensional symbols on the other (lines, arrows, etc.), inclusion is impossible, and touching or overlap are no real options. Wrapping the text in a box and connecting it to the associated symbols is an option in this case, but would make the graphical representation denser than needed.

In AMBER therefore, we mostly use inclusion, overlap, and connection for association. For instance, names of model elements are included in the symbol.

Of course, grouping is another way of association. AMBER therefore includes grouping concepts such as blocks and actors, with appropriate visualisation.

Analysis

Second, the *analysis* of business processes should be accompanied with accessible representations of the analysis results. Analysis results can have many forms, such as tables, mathematical expressions, numbers, or parts of the business process itself. We will restrict our discussion to those analysis results that are business processes themselves or parts thereof. We call these analysis features *views*. Views are used for hiding of highlighting specific elements of business process models or for presenting a business process model from a specific perspective. They are discussed in section 5. Hence, we will not go into details of the analysis possibilities offered, nor how they should be used by a business architect.

Planning and migration

In *planning and migration,* a business architect should be able to visualise entire trajectories of business process models. Additionally, Testbed should support the guidance of the implementation of the new process to some extent. This includes generating functional specifications for system development, writing working instructions for employees, training and educating people of the organisation, and exporting business processes to for instance workflow management systems.

Communication

In any step in this process, a business architect should be supported in his or her *communication* with specific groups inside the company or companies involved. As these groups will have different goals and different backgrounds and hence different visualisation needs, visualisation has to offer ways of tuning business process visualisation to their specifics. For instance, company management may have little interest in elaborate presentations of all business process details; to them, black-box abstractions may be shown. On the other hand, when for instance translating business process models into operational instructions for shop floor employees, many details may have to be included.

3. AMBER — a graphical business process language

This section introduces the business process design language AMBER and its graphical representation. A separate paper [2] is dedicated to a more elaborate presentation of the language. When introducing a language concept, we mention its representation between square brackets, as follows: action [circle].

AMBER's core recognises three aspect domains:

– the *actor* domain, describing the resources (deployed for) carrying out business processes; and
– the *behaviour* domain, describing the activities happening in a business process;
– the *item* domain, describing the items (forms, files, databases, etc.) handled in business processes.

AMBER models are not depicted by one huge representation. The language includes several concepts for decomposing models into smaller parts, such as blocks and components. Besides, model representations can be split up into diagrams. Each diagram is an actor diagram, a behaviour diagram or an item diagram. The separation between diagrams however has no formal semantics; it serves no other purpose than to split the representation.

3.1 The actor domain

The basic concept in the actor domain is the *actor* [octagon]. It designates a function, role, organisational unit, person, or system (used for) carrying out a business process. Actors may be nested as well as replicated [shadowed oval]. Actors generally have *interaction points* [oval], which are physical or logical locations at which the actor may interact with its environment. *Interaction-point relations* [(multi)lines] connect interaction points. Any interaction point can be involved in more than one relation. In addition, a relation may connect more than two interaction points. Figure 3 shows a typical actor model.

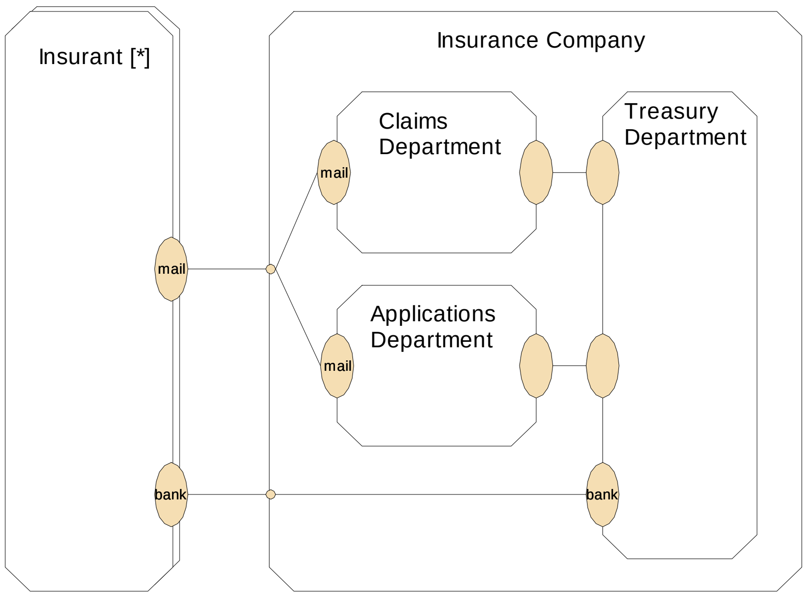


Figure 3 — A typical actor model.

For actors containing other actors, a nested representation is preferred over a tree-like representation. This enables a better representation of the relation between the interaction points of the constituents on the one hand with the interaction points of the containing actor at the other. However, Testbed Studio provides a view, in which the tree-like representation can be automatically generated (see 5.3).

3.2 The behaviour domain

The basic concept in the behaviour domain is the *action* [circle]. It models a unit of activity in business processes. An action can only happen when its *enabling condition* [arrows] is satisfied. These conditions are formulated in terms of other actions having occurred yet, or not. The most simple is the enabling relation [simple arrow]. When included between actions a and b, it models that b cannot happen but after a has finished. Its mirror image is the disabling relation [arrow with crossing line], modelling that b cannot happen any more, once a has occurred. Enabling relations can be composed using splits [diamond] and joins [little rectangle]. An and-split [filled diamond] models a parallel fork, an or-split [empty diamond] models exclusive choice, an and-join [filled rectangle] synchronises, and an or-join [empty rectangle] enables only when enabled by at least one enabler. See Figure 5.

Additional *constraints* [text] (typically, on attribute values of preceding actions) can be used to further restrict the enabling relation. Actions, interactions, and blocks may be *replicated* [shadow]. *Iteration* is used for modelling repeated behaviour [double edge and double-headed arrow].

A special kind of action is a *trigger* [frayed semicircle]. Triggers are like actions, except that they are always immediately enabled. Actions can be grouped in *blocks* [rounded rectangle]. Blocks can be nested. There are two ways to separate a block from its environment. One is *between* actions; the other is *inside* actions. When a block separates behaviour between actions, a causality relation is cut. An *entry* [inward triangle] or *exit* [outward triangle], depending on the direction of the causality relation indicates the cutting point (at the block's edge). This is typically used for phasing different parts of a business process. When a block separates behaviour inside actions, the action is divided in a number of *interactions* [semicircle]. *Interaction relations* [(multi)lines] are like interaction-point relations in the actor domain. An interaction can only happen simultaneously with all of its related interactions, which must therefore all be enabled. This type of structuring is typically used for modelling interaction between actors. Phased and interaction-based structuring may be arbitrarily mixed. Figure 4 presents a typical behaviour model.

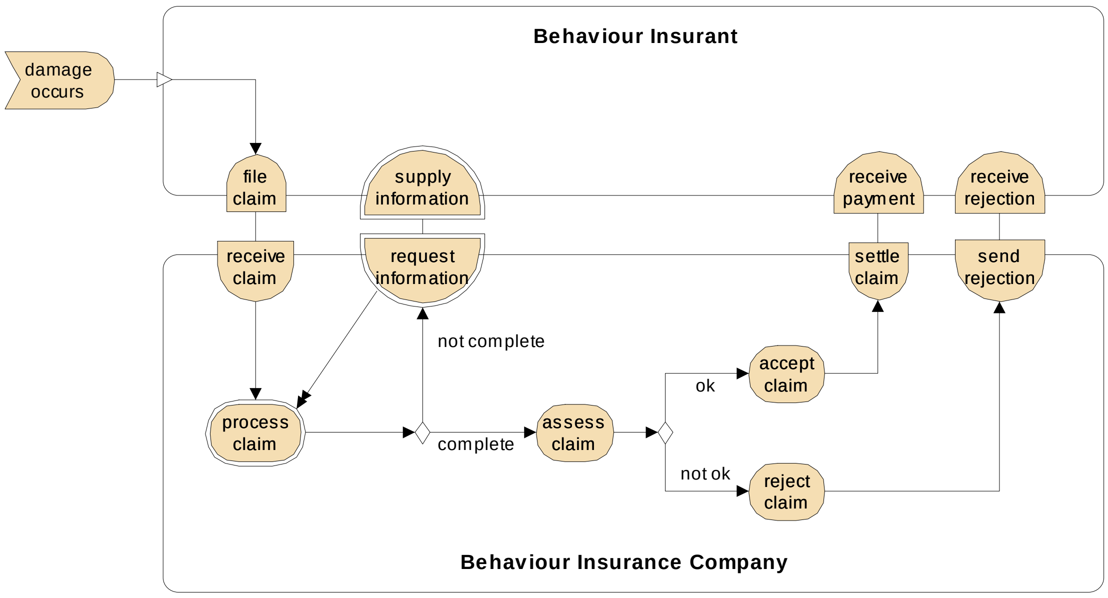


Figure 4 — A typical behaviour model.

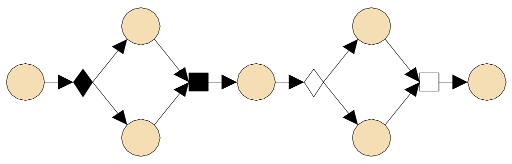


Figure 5 — From left to right: and-split, and-join, or-split, or-join.

3.3 The item domain

The item domain models the *items* [rhomboid] (files, forms, data) on which (inter)actions are performed by the actors. In the actor and behaviour models, items can be included and coupled to the various elements of these models. In the actor domain, items are coupled to interaction-point relations, indicating that in the interaction-point relation involved, the indicated item is used.

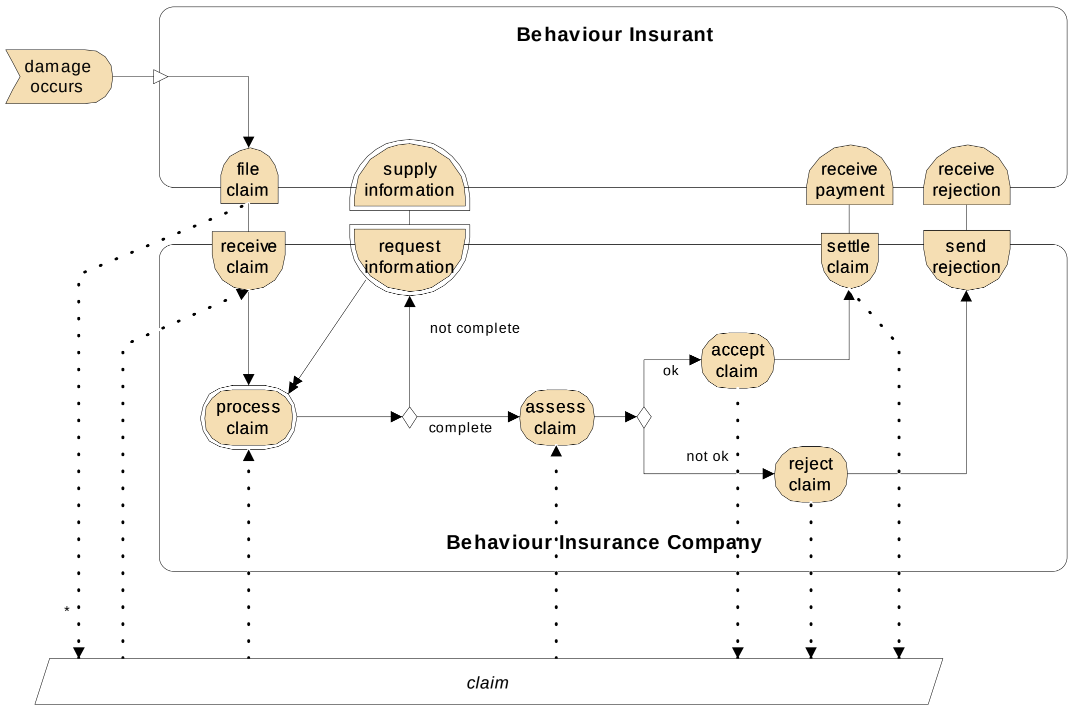


Figure 6 — Items in a behaviour model.

Coupling items to elements of behaviour models is different in two ways. First, the items are coupled to actions and interactions, instead of to interaction relations. Second, item coupling in behaviour models distinguishes between five modes: create, read, write, read/write, delete [Figure 7]. This mode indicates what type of action is performed on the item involved. Figure 6 shows how items can be used in a behaviour model. Usage in an actor model is similar.

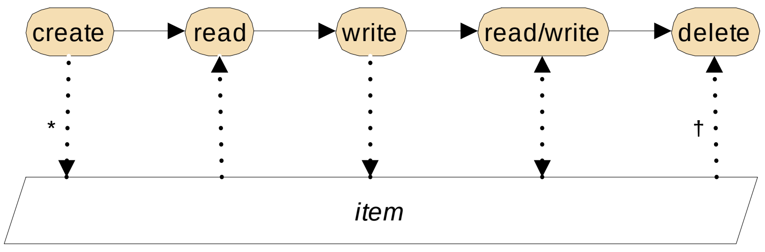


Figure 7 — Items in behaviour models.

Currently, an item definition language is being added to AMBER. This language is based on a subset of the Unified Modelling Language UML [7]. It will provide structuring of, operations on, and relations between items.

3.4 Components

Where actors and blocks are concepts for structuring individual business-process models, AMBER also contains a concept for inter-relating different business process models: components. Components are (partial) business process models, which exist on their own and may be stored in a repository. Each component contains at most one of each of the following: an actor model, a behaviour model, and an item list. Components can be used in other models, and can themselves carry gaps, in which other components can be fitted in. In this way, component re-use and concurrent business-process design is enabled. Figure 8 shows how *component gaps* [grey box with an underlined name] are visualised in a behaviour model: ‘Behaviour Insurance Company’ is a component gap. By default, the component contents are hidden, but they can be made visible as well.

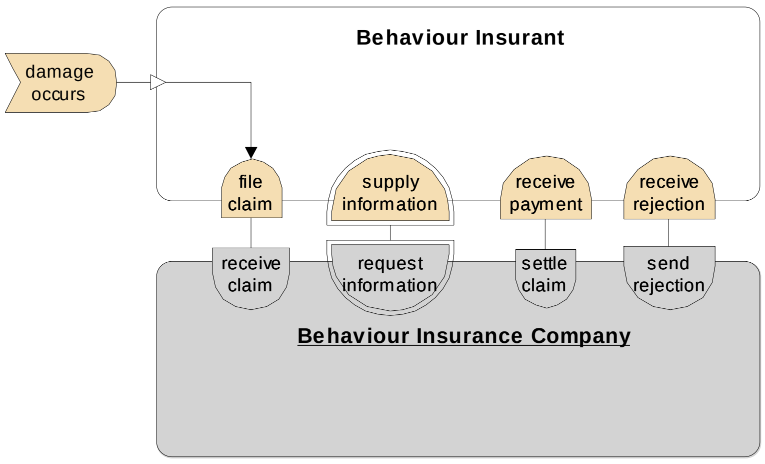


Figure 8 — Component in a behaviour model.

3.5 Attributes and profiles

As pointed out, actors are (used for) carrying out business processes. Hence, elements of the actor domain should be coupled to elements of the behaviour domain. This is done by associating attributes with blocks, actions, and interactions. These attributes are references to actors. There may be several of these attributes, because actors may play different roles. In some cases, actors are the resources used in business processes, whereas in other cases, actors may be the ones responsible for certain business processes. An option would have been to introduce several actor concepts and representations for each of these roles. This however would affect the conciseness of the language. Yet, it is possible to specifically visualise a single role by means of views (see section 5).

In fact, any language concept may carry attributes, for any purpose. Attributes are grouped in so-called profiles. Profiles allow for specialising concepts for specific purposes, such as for domain-specific modelling (modelling logistics processes, for instance) or for specific analysis purposes (lead-time analysis, for instance, requires specific attributes with actions).

4. Representation conventions

As pointed out above, many visualisation requirements have been incorporated into the design of the AMBER language. Notwithstanding their helpfulness in creating clear models, a number of visualisation features however should not be entrenched in the language, because it is undesirable to force the business architect to apply them. In [6], where these features are called 'secondary notation', their importance and effects are discussed. We call them representation *conventions*.

Representation conventions can be applied to increase the ease of understanding models. Especially for the experienced user, they may provide useful clues for the meaning of a model. Using conventions does not influence the formal meaning of the model. Typical conventions encountered in textual programming are naming conventions and indentations conventions for clarifying the nesting structure of the code.

Conventions cannot be applied very strictly in all situations. Sometimes conventions conflict and sometimes it is unavoidable to violate a convention. Below, we present a number of useful conventions. Though included in the handbook accompanying Testbed Studio, the use of conventions is optional.

From left to right

By letting enabling relations point from left to right, the time dimension in behaviour models matches the natural reading direction in Western cultures. Of course, this convention has to be violated in case of a loop.

Symmetry

Symmetry can be used to suggest or stress similarities between parts of the model (Figure 9).

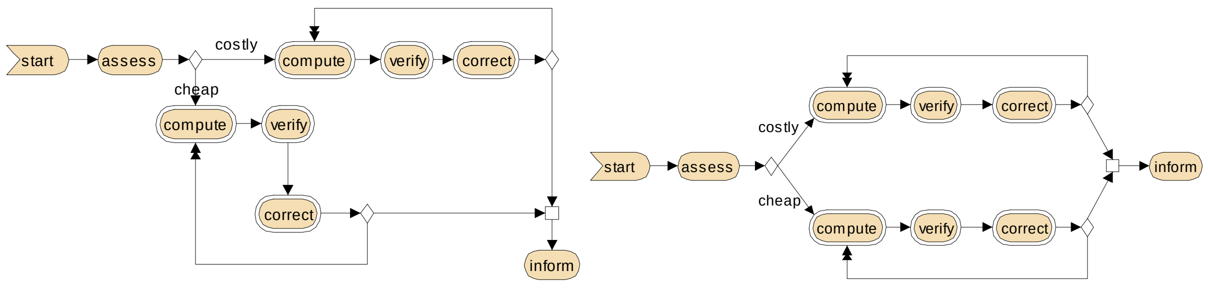


Figure 9 — Symmetry in business model visualisation.

Colour

Another way of indicating that certain elements have something in common is using the same colour. As pointed out, colour has no formal meaning in AMBER. For instance, actions performed by the same resources can be given the same colour as the representation of the resource. In Testbed Studio, colour views can be based on several criteria (attributes), such as the actor involved.

Naming

Naming conventions can be used for indicating the kind of element, such as verbs for actions and nouns for resources.

Suggestion of time duration

By exploiting the space between actions, time duration can be suggested. The left model in Figure 10 suggests that first *a* and *b* take place and *c* takes place only after a rather long time. The right one, however, suggests that a, b and c take place with more or less equal time slots in between.

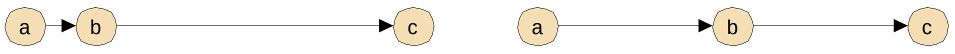


Figure 10 — Suggestion of time.

Normal versus exceptional cases

In order to reduce the complexity of a model, it is useful to make a clear distinction between the normal proceedings and exceptions thereto. This can for instance be realised by presenting the normal activities within a process at a straight line, while placing the exceptional activities above or below that line. See Figure 11.

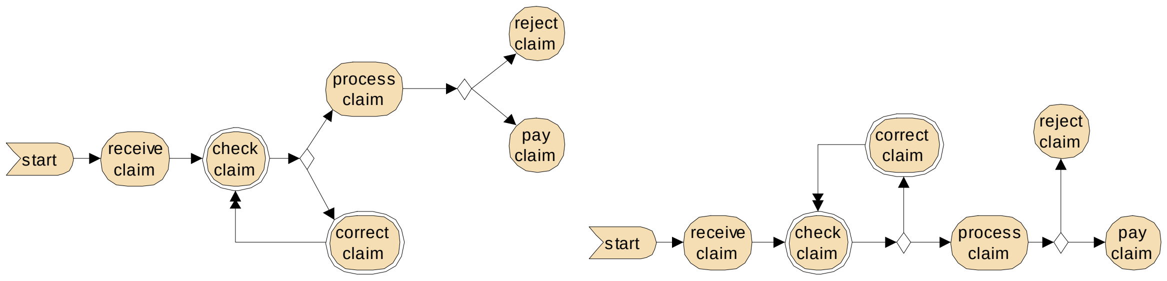


Figure 11 — Visually separating normal course of action from exceptions.

Avoiding crossing lines

Avoiding crossing lines can increase the readability of a model. In case of a crossing line, the user may have to spend extra time to figure out in what direction each of the lines continues.

5. Views of business process models

5.1 Four types of views

Different users and different goals require different representations or *views* of a model. Many aspects and elements of a model need not be visible in all views of that model. If we would, for example, textually display all profile attribute values (see 3.5) of a complex model, the result would be unreadable. We prefer to selectively show only those aspects of a model that fit a specific purpose.

Views might use symbols, structure, colour, line style, and other features to emphasise specific aspects of a model. Having to draw all these different types of views by hand would be a rather demanding task, which would unnecessarily duplicate much of the work invested in the basic model.

Therefore, Testbed Studio supports automatic generation of different types of views:

- *graphical views*, which highlight elements of a model, by using colour, line style, and other graphical attributes.
- *textual views*, which display non-graphical information (typically, attribute values) contained in a model, by labelling model elements with text.
- *structural views*, which restructure a models representation, such as an organigram depicting the nesting hierarchy of an organisation's departments as a tree structure, or process lanes.
- *simulation*, which is a dynamic representation of a business process.

5.2 Hiding and showing model elements

An important application of views concerns the selective hiding of parts of a model, for instance for the purpose of presenting high-level business models for management. Given a large, complicated model, it is important that the user can choose which parts of that model he or she wants to see.

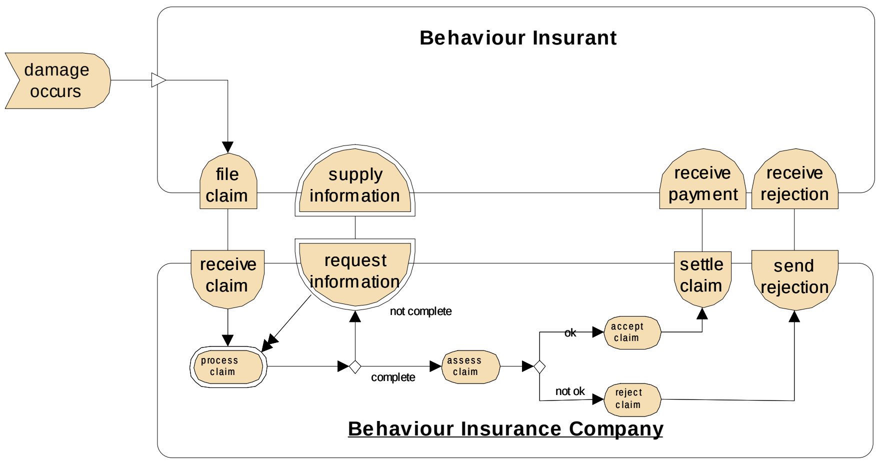


Figure 12 — Scaled block contents.

If, for example, a behaviour model contains a deeply nested block structure, a user might want to see only the top-level blocks. To this end, Testbed Studio offers an implode/explode view, which can hide or show the contents of selected actors and blocks. An imploded actor or block has its name underlined. It can be exploded in a separate diagram; its contents may also be shown in the original model, if necessary scaled to fit a resized actor or block, as illustrated in Figure 12. The same view can be applied to the contents of components (see Section 3.4).

5.3 Organigrams

An organigram is a structural view, helpful in the analysis of the hierarchical structure of an organisation, omitting other aspects such as resource capacities. Testbed Studio can automatically generate organigrams and other hierarchical diagrams. Organigrams are tree-formed representations of nested actor models. For instance, the organigram of Figure 13 corresponds to the nested organisation structure of Figure 3.

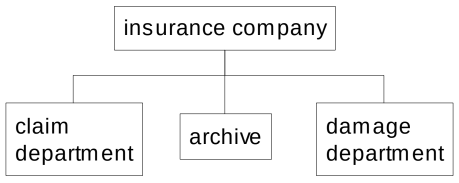


Figure 13 — Organigram of an insurance company.

5.4 Viewing attributes

Using attribute views, the information contained in the attributes of model elements can be shown in several ways. An example is the assignment of behaviour to actors. As we have seen in Section 3, behaviour elements are assigned to one or more actors that carry out this behaviour.

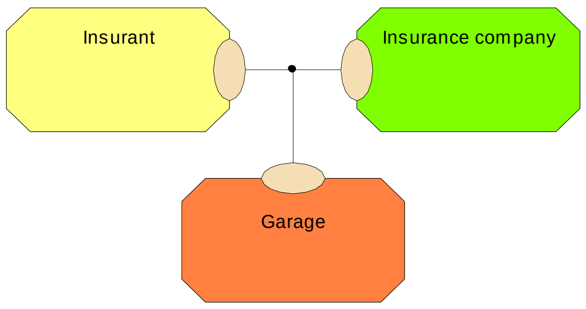


Figure 14 — Actor model.

Consider the actor model of Figure 14, in which the colours have been adjusted manually. To represent the actor information in a behaviour model, we can use different types of views:

- The behaviour elements can be given text labels with their actors.
- The behaviour elements can be given the colour of the corresponding actors (Figure 15). Some actions have multiple actors; they remain white. Another option would have been to use all colours at once, but this would have yielded very colourful models.
- The behaviour model can be structured according to the assignment of behaviour to actors: the behaviour assigned to a single actor is shown as a horizontal band in the behaviour model (Figure 16). Actions with multiple actors are split in connected interactions, one for each actor.

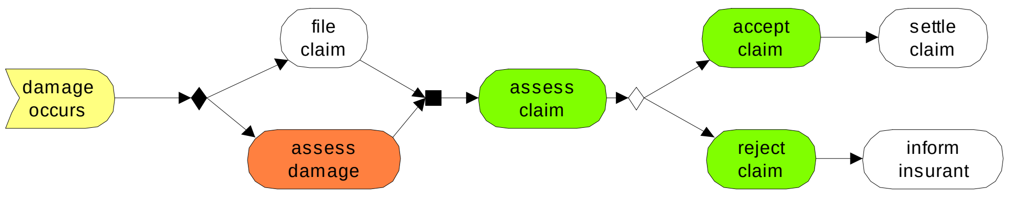


Figure 15 — Actions coloured according to their assignment to actors.

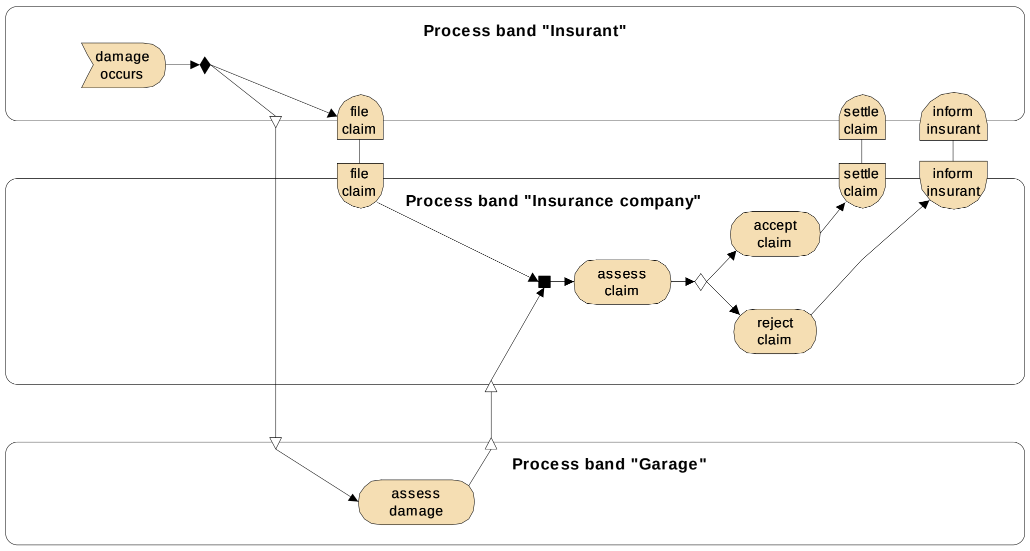


Figure 16 — Behaviour structured according to the assignment to actors.

5.5 Analysis views

Views can also be used to represent results of the analysis of a model. Testbed Studio makes extensive use of this, in several types of analysis:

- Quantitative analysis computes several quantitative performance and cost measures. Based upon these measures, the tool can, for example, highlight the critical path, i.e., the actions in the model that are primarily responsible for the completion time of a process.
- Functional analysis uses model checking [4] to analyse functional properties of a model, such as “is action *a* always followed by action *b*?”, or “can actions *a*, *b*, and *c* occur together?” If a counterexample of such a property exists, it is graphically represented using colour in the original model.
- Precedence analysis colours the actions that necessarily or possibly precede a given action.

– Data-flow analysis highlights the paths by which information, represented by items, flows through the behaviour model.

5.6 Simulation

A model can be simulated (or animated) by Testbed Studio. Different colours are used to signify whether an action is enabled (green), is executing (orange), or has executed (grey). See Figure 17.

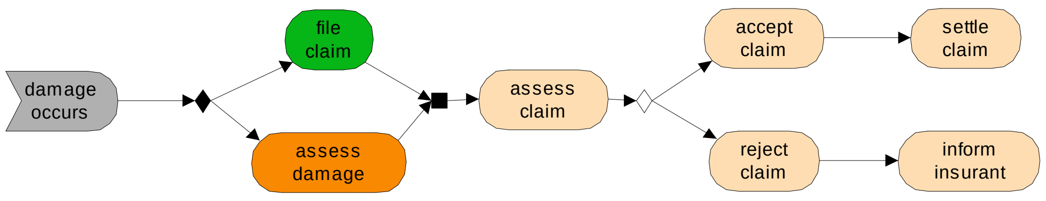


Figure 17 — 'damage occurs' has executed, 'assess damage' is enabled, 'file claim' is executing.

6. Adherence to requirements

In this section, we discuss the way in which AMBER's visualisation features, as discussed in sections 3–5, match the major requirements of section 2.

Unambiguity

A major, yet simple decision for avoiding ambiguity is taking different symbols for different language concepts. Notice however, that (multi)lines are used for both interaction-point relations and interaction relations. Their context however easily disambiguates here. Interaction point relations occur in actor diagrams only, interaction relations in behaviour diagrams only.

It is also possible to let the immediate context decide how to interpret arrows, that is, as part of an enabling relations or as an association between items and (inter)actions. However, because both can occur in the same diagram, we decided to have item associations denoted by dotted arrows.

In some cases, a combination of symbols may imply an ambiguity risk. This occurs in AMBER when lines cross. Therefore, crossing lines carry a tiny circle when semantically connected and no tiny circle when not.

Organisation

AMBER offers many ways of visually organisation models. The first is the separation between diagrams. Conventions are another option for visual organisation, although they are mere suggestions instead of language features. AMBER's views however offer the most powerful way of visually arranging business process models.

Economy

AMBER has been given simple representations for each of its concepts. Visual orthogonality (see below) however also helps in designing economical representations, because orthogonality allows for using as few visual characteristics as possible to express a wide range of meanings.

Orthogonality

Orthogonality is used extensively throughout. First, the actor domain and the behaviour domain include analogous concepts (interaction points versus interactions, actors versus blocks). The representation of these analogous concepts is designed analogously.

Replication and iteration are indicated by visual variants on the basic symbols. Throughout, replication is indicated by a shadow. Iteration is indicated by double edges and double-headed arrows.

Interactions are denoted by semicircles, in order to evoke the idea that multiple interactions together form one action.

The orthogonality principle is also applied in

- the symbols for splits (diamonds) and joins (squares). In either case, the disjunctive version is unfilled and the conjunctive version is filled (Figure 5).
- choosing for the two-sided arrow for read/write, the symbol for creation (which is a special case of writing), and deletion (which is the inverse of creation).

External consistency

This is a difficult requirement to meet, because the business architect’s expectations and experience are largely implicit and may vary between individuals. There is no explicit evidence that AMBER is externally consistent.

Yet, all results of the Testbed project have been and still are applied in real-world settings in large information-intensive organisations. In a number of cycles, the AMBER language, including its visualisation features, has been adjusted to remarks and wishes of business architects. However, there has so far been no systematic evaluation by users, dedicated to the visual aspects of the language and the tool.

Size and orientation

Actions and interactions are stretched when needed to include longer names. This however may cause differences in size, suggesting difference in importance. Their formal meaning however does not change.

The representation choices for splits (diamonds) and joins (squares) holds an ambiguity risk, since rotated squares may be seen as diamonds. This however, is solved by letting the arrows apply only at the angles of diamonds and the edges of squares.

7. Further research

Further work in the area of visualisation remains to be done in the following areas:

- development of a generic colour view, where any attribute of a model element can be used as the basis for a colour view. This is a generalisation of the specific colour views already available.
- visualisation of data and items. Along with the inclusion of item modelling in Testbed Studio, a visual data definition language will be offered. The data manipulation language will be textual. UML will serve as the example.
- visualisation of layered models. Currently, implode and explode views are offered for individual blocks and actors. In layered models, implosion and explosion might be offered for all blocks or actors at a certain nesting level at once.
- specific visualisation of different model types and abstraction levels.

- visualisation of sophisticated analyses. In the future, additional advanced analysis capabilities will be included in Testbed Studio, such as stochastic simulation. The more complex analyses are performed, the more attention should be paid to accessible visualisation of their results.
- visualisation of domain-specific concepts or visualisation for specific user groups.

References

[1] F.P.M. Biemans, M.M. Lankhorst, W.B. Teeuw, and R.G. van de Wetering (1999), Mastering the complexity of business process design, Submitted to: Systems Research and Behavioral Science, Wiley.

[2] H. Eertink, W.P.M. Janssen, P.H.W.M. Oude Luttighuis, W. Teeuw & C.A. Vissers (1999) A Business Process Design Language, In: 1999 World Conference on Formal Methods (FM'99, Toulouse, France, 20 – 24 September, 1999).

[3] H.M. Franken & W.P.M. Janssen (1998). Get a grip on changing business processes, *Knowledge & Process Management* Winter 1998; Wiley.

[4] W.P.M. Janssen, R. Mateescu, S. Mauw & J. Springintveld (1998) Verifying Business Processes using SPIN. In: G. Holzman, E. Najm E & A. Serhrouchni (eds) Proceedings of the 4th International SPIN Workshop, Ecole Nationale Superieure des Telecommunications: Paris, France; Report ENST 98 S 002; pp. 21 – 36.

[5] A. Marcus (1997). Graphical User Interfaces, Chapter 19 in: Handbook of Human-Computer Interaction, Second edition, Elsevier Science, pp. 423 – 440.

[6] M. Petre (1995). Why Looking Isn’t Always Seeing: Readership Skills and Graphical Programming. Communications of the ACM 38(6), 33 – 44.

[7] Rational Software Corporation. (1997). Unified Modeling Language, Version 1.0. http://www.rational.com/ot/uml/1.0/index.html.

[8] C. Szyperski (1998). Different programming methods for different programmers, Section 10.1 in: Component Software — Beyond Object-Oriented Programming, Addison Wesley, 1998 (reprint), pp. 145 – 147.

Acknowledgement

The authors wish to acknowledge all participants in and sponsors of the Testbed project, a 120 man-year research initiative to develop languages, methods and tools for business process design in the financial sectors. The Testbed consortium consists of ABP, the Dutch Tax Department, ING Group, IBM and the Telematics Institute (The Netherlands). It subcontracts research projects to several research institutes. The project is further supported by the Dutch Ministry of Economic Affairs. We further acknowledge the Architecture Group of the University of Twente, Enschede, the Netherlands (Prof. Chris A. Vissers) which has developed Testbed's architectural modeling technique.

Furthermore, the authors thank Ynze van Houten, Frank Biemans, and Henry Franken from the Telematics Institute for their useful comments on earlier versions of this paper.

List of Figures