

What Makes Industries Believe in Formal Methods

Chris A. Vissers^{a,b}, Marten van Sinderen^b, Luís Ferreira Pires^b

^aTelematics Research Centre, P.O. Box 217, 7500 AE Enschede, the Netherlands

^bTele-Informatics and Open Systems Group, University of Twente,
P.O. Box 217, 7500 AE Enschede, the Netherlands

The introduction of formal methods in the design and development departments of an industrial company has far reaching and long lasting consequences. In fact it changes the whole environment of methods, tools and skills that determine the design culture of that company. A decision to replace current design practice by formal methods, therefore, appears a vital one and is not lightly taken. The past has shown that efforts to introduce formal methods in industry has faced a lot of controversy and opposition at various hierarchical levels in companies, resulting in a marginal spread of such methods. This paper revisits the requirements for formal description techniques and identifies some critical success and inhibiting factors associated with the introduction of formal methods in the industrial practice. One of the inhibiting factors is the often encountered lack of appropriateness of the formal model to express and manipulate the design concerns that determine the world of the engineer. This factor motivated our research in the area of architectural and implementation design concepts. The last two sections of this paper report on some results of this research.

1 Introduction

Design and Production Culture

Industries aim at continuity and making profit. These goals are achieved by considering several important factors such as product range, marketing approach, stock control, productivity and competitiveness. The latter two factors determine the capability to produce the ‘best’ possible products in the shortest possible time and at the lowest possible cost. Productivity and competitiveness are largely determined by the methods, procedures, tools and skills necessary for the design and production of products, and the way the people involved in them are able to handle these facilities and faculties in their daily work. They determine a vital aspect of the specific organization and working practices of an industry and form its specific design and production environment. This environment is carefully cultivated and at the same time screened off from the competition, determining a specific (design and production) culture of that industry. Figure 1 depicts some elements of a design culture, and their relationship with a specific design instance ([13]).

Any industry with a well understood self-interest will continuously search for new and better methods, procedures, tools and skills to optimize its design and production environment, aiming at improving its productivity and competitiveness and with the ultimate goal to safeguard its continuity and profit. Being a vital component of its survival strategy, however, this also implies

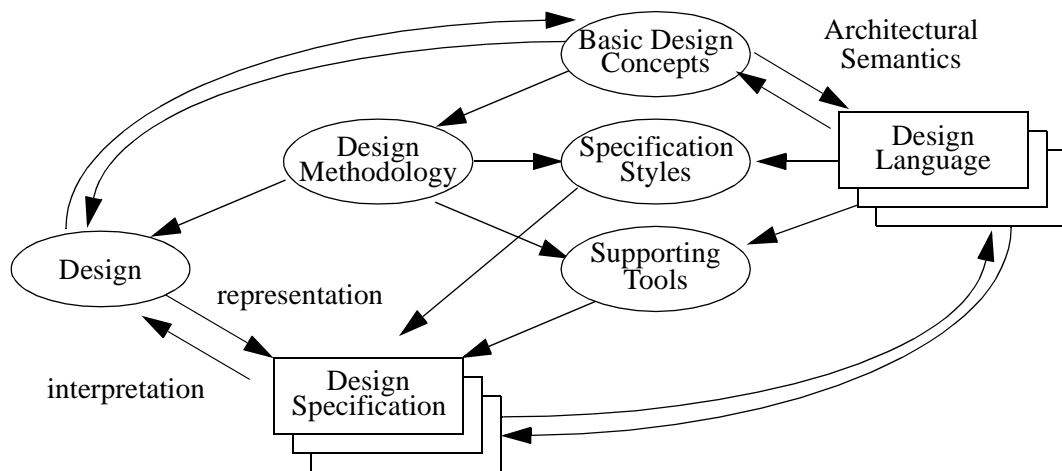


Figure 1: Elements of a Design Culture

that an industry will consider modifications to its design and production environment with the greatest care so as not to put its productivity and competitiveness at stake. Only if substantial improvements to productivity and competitiveness can be *guaranteed* an industry will seriously consider new methods, and will afford the investments in money and time to introduce them into its daily practice.

Consequently, methods, procedures, tools and skills are seen as *means to achieve a goal, not goals in themselves*.

Formal Methods

In the last decade a substantial sympathy has been raised for the development and application of Formal Description Techniques (FDTs). These developments were to a large extent triggered by the need to overcome the so called *software crisis*: the inability to control the complexity and correctness of large software designs. Such lack of control is particularly harmful for the development of large and modern distributed systems where hidden errors in the high level design and specification may cause great difficulties in their operation and maintenance.

FDTs originally aimed at controlling the unambiguity and correctness, not necessarily the complexity, of designs. As such their development and application was particularly fostered by ISO and CCITT since these standardization bodies had a particular interest in bringing out their specifications of standards and recommendations for (e.g. open systems) protocols and services in an as unambiguous, clear and concise way as possible. These standardization bodies even decided themselves to develop international standards and recommendations for FDTs.

A corresponding sympathy was raised in (inter)national research programs like ESPRIT, RACE and Alvey. Large FDT oriented research projects, like SEDOS, METEOR and SPECS, were launched. SEDOS focused on the FDTs LOTOS and Estelle and contributed substantially to the work in ISO and CCITT. SEDOS appeared an important factor in progressing LOTOS and Estelle to the status of international standard. SEDOS also contributed substantially towards the development of formal specifications of ISO standards in these FDTs like the ones for the Transport and Session layers [6], [7], [5], [4]. Successor projects like Lotosphere and Estelle

Demonstrator were capable of building on top of SEDOS and developed quite sophisticated design and tool environments around these FDTs.

As a result one can observe that a lot of skills, time and money have been invested in the last decade in the development of advanced *Formal Methods* (FMs), i.e. the FDT language definitions, related design methodologies and design support tools. By involving large industrial partners in these efforts serious attempts have been made to advance the uptake of FMs by industry.

At the same time an abundance of FMs has been developed and reported in conferences like PSTV, FORTE, SDL Forum, FME and AMAST.

As said above, any self-respecting industry will not lightly take the decision to modify its design and production practice by new methods, let alone replace it completely and overnight. However, given the continuous search for new and better methods by industry, given the primary aim of FMs to provide new and better methods, and given the current availability of quite advanced and sophisticated methods based on standard FDTs, one would expect a broad scale industrial interest and uptake of these FMs. To what extent is this uptake underway? Is the picture indeed much different from the one of ten years ago?

State of the Art

Observing the state of the art in this respect shows a discouraging picture. In contrast to the broad scale interest in Universities, we see at the best that some large industries have shown a willingness to consider, apply, and even extend FMs, however, generally on an experimental basis only. In certain institutions indeed some specific error critical projects are carried out (see for example [2]) but the number of these institutions is small. Serious product developments do not take place on a broad industrial scale. At any case we can observe a situation which is far away from the large scale breakthrough of FMs.

Moreover one can observe a widespread indifference with respect to FMs. In CEC funded programs, like ESPRIT for example, the chances of getting projects accepted that aim at developing FMs based enabling technologies is marginal nowadays. The number of FDT oriented tools that are installed in industry is limited. Several industrial circles simply and openly reject the usefulness of FMs. FMs are oversold, they say: “they have promised too much, but appear not useful in practice”. In ISO/IEC-JTC1, originally a driving force in their development, we currently see efforts to abandon the use of FDTs ([8]).

Several arguments may be tossed to excuse the current situation. One may argue, while referring to structured programming for example, that the uptake by industry of advanced techniques always takes a ten to twenty years learning curve. One may also argue that the introduction of FMs, through their rigidity and required strict obedience to the semantics of an FDT, will have a more than usual impact on all aspects of the design culture of an industry, and thus will naturally face resistance of those who do not have the necessary background. Maybe FMs will only be introduced through the influx of formally trained students.

Is the industry ignoring its chances? Do current FMs not really support productivity and competitiveness? As ever, any situation is never black and white, and the above excuses to a certain extent hold. However, the reasons for the reluctance and critical position of industry hold to the same extent. Facts that should be taken seriously! If not, we may easily jeopardize the cause of FMs and wind up in, what we may call, a *Formal Methods Crisis*, i.e. the inability of FMs to control the correctness and complexity of large software designs.

The question we have to ask ourselves is: what requirements should we really put to FMs, and what has to be improved in order to make industries believe in them. In order to tackle this

question, it is useful to see what lessons we can learn from the development of the FMs related to standard FDTs.

2 Original Requirements for FDTs

The interest of standardization bodies in the development of unambiguous specifications of standards led in ISO in 1979 to the installation of the ISO/TC97/SC16/WG1/FDT group. The objectives and requirements for FDTs were formulated in Annex E of the OSI Reference Model ([3]). In summary, they should provide a basis for:

- the development of unambiguous, clear, and concise specifications,
- the verification of specifications,
- the functional analysis of specifications,
- the development of implementations from a specification,
- the determination that such implementations conform to their specification.

Within CCITT a similar group already worked for several years on the development of SDL.

At first glance it seems that the above requirements cover all relevant concerns related to FMs development. But, without disputing the relevance of these requirements, we can at the same time make the following observations following the history of the different FMs developments within CCITT and ISO:

1. SDL (version 1980 with graphical notation only) was considered too much hardware and implementation oriented by ISO, who was particularly interested in the software oriented and the implementation independent specification of OSI standards. The different views on implementation independence in fact pre-occupied the orientation of the different FDT groups. Comparing the resulting standards and recommendations against this requirement we can observe that LOTOS is more suited for abstract specifications, whereas Estelle and SDL are closer to implementation oriented specifications. As a net result LOTOS provides better specifications of OSI standards, whereas specifications in Estelle and SDL, once available, are much easier to implement. The question can be asked why these different FDT groups made such vastly different interpretations of the level of abstraction at which a specification of the same technical object should be provided.
2. LOTOS on the one hand, and SDL and Estelle on the other hand, are based on completely different formal models. LOTOS is based on an (asynchronous) labeled transition system model, whereas models are interconnected via synchronous interaction. SDL and Estelle are based on a (synchronous) finite state machine model, whereas models are interconnected via asynchronous (infinite queues) interaction. Consequently they have completely different views and modelling power for important architectural notions such as the OSI Service boundary ([14]). The question can be asked why these different FDT groups made such a vastly different interpretation of important architectural notions with such drastically different consequences for the choice of formal models and resulting modelling power?
3. LOTOS, SDL, and Estelle have shown different degrees of success in producing specifications of OSI standards. Initially SDL and Estelle appeared quite unsuccessful in producing Formal Descriptions (FDs) of OSI standards. On the other hand LOTOS should not be too proud of its specifications of Transport and Session layer standards. The development of the Transport Protocol Technical Report took about 8(!) years, and of the Session Service and Protocol Technical reports about 5 years. Moreover, these reports are not used, most probably not error free, and, in case of the Session layer reports, no longer up to date. In any case

one can observe that LOTOS and Estelle specifications that were derived from the same informal specification are vastly different from each other. Consequently the implementations will also be vastly different from each other. The question can then be asked why there should be such drastic differences between specifications and resulting implementations of the same technical object?

As a side-remark we note that CCITT and ISO have different views on how the application of FDTs should be guided. CCITT is dealing with recommendations for PTTs, and PTTs often require that manufacturers produce specifications of their products in SDL, or they produce specifications in SDL themselves. In ISO there is complete freedom to use, or not to use, FDTs. As a consequence SDL is spread much wider than LOTOS and Estelle.

The fact that these standard FDTs have quite different interpretations about what to model and how to model, leading to completely different architectural interpretations, specifications, and implementations of the very same technical object is bewildering. Corresponding differences in interpretation and expression can hardly be found in other engineering disciplines like electrical or mechanical engineering. If we also consider the overwhelming amount of non-standardized FDTs developed everywhere we observe even wider differences. What is especially worrying is the lack of interest among formalists to find out the right interpretations. Everyone maintains the status quo. A well known example is the modelling of the execution of a service primitive at the service boundary: in LOTOS this is done synchronously by the interaction concept, and in SDL and Estelle this is done asynchronously by a message exchange via an infinite queue.

Given these large divergencies it seems fair to raise the following questions: What are actually the objectives and requirements for FMs development for our field of application? Are these requirements formulated precise and clear enough? Do these requirements provide enough guidance for FMs development? Does the FMs community agree on these requirements? Indeed it seems as if the FMs community should first agree on what the requirements of the engineering of (distributed) information systems really are before elaborating FMs for this field. Below we want to contribute to a discussion on this topic by considering some requirements and reasoning about them, while using in particular our experience with the developments around LOTOS.

3 Requirements Revisited

Without ignoring or diminishing the original objectives and requirements of FMs, and without ignoring many other, more detailed, requirements that one can put forward for FM developments, we think that the following is a set of major requirements that should be observed in order to obtain a more successful uptake of FMs in industry: appropriateness, design methodology, design support tools, migration path, limited number of methods, and educational material. We elaborate each of these requirements below.

3.1 Appropriateness

By appropriateness we mean the measure in which the formal model underlying the development of a set of FMs suits the purpose of its application area. Like in any other technical science, such as mechanical or electrical engineering, the choice of formal model should be determined by the needs of the application area in order to be applicable at all. In our case this appli-

cation area is the specification, implementation and production, shortly the development, of (distributed) information systems.

We believe that, in order to be appropriate, the formal model should provide a complete set of correct abstractions of relevant engineering concepts which are needed to design distributed systems.

3.1.1 Relevant Engineering Concepts

The question what are relevant engineering concepts that underly the development of a formal model is probably the most difficult question to answer. This may very well be the reason why we have such vast differences in formal models. Yet this question is most vital and *must* be answered.

Relevant engineering concepts provided by the OSI Reference Model are for example *unit of behaviour*, like a service provider or a protocol entity, *interaction point*, like a service access point, *interaction point identifier*, like a service access point address or a connection endpoint identifier, *unit of interaction*, like a service primitive, *internal data unit*, like a protocol data unit, etc. But also concepts such as *causality*, *concurrency*, *conflict*, *timing*, *observable behaviour*, *internal behaviour*, *behaviour composition constructs*, which allow the structuring of complex behaviours through compositions of more elementary behaviour, are important engineering concepts ([9]). Addressing such concepts in the OSI Service Conventions document¹ has learned that OSI architects themselves can get involved in quite harsh debates about the correct interpretation of relevant engineering concepts, such as for example: “the relevance of the notion of direction of a service primitive”.

An important starting point for the design of LOTOS was support for the representation of designs in an implementation independent way. A technical object like a service provider or a protocol entity in LOTOS is therefore defined in terms of its *observable behaviour*. This led to the notion of the interaction concept with multi-party synchronization, which is probably the most important design concept in LOTOS. At the same time, however, another important requirement in the LOTOS development was overlooked: the necessity to support the whole implementation trajectory, comprising implementation dependent specification and implementation notions. This requires among others the possibility of allowing the representation of *internal behaviour*. The Lotosphere project could not restore this omission, since it is a lack of the LOTOS formal model. In our research this concern has led to the definition of the *action* concept (see Sections 4 and 5). For SDL and Estelle on the other hand, too much emphasis was placed on implementation dependent specification.

We stress that engineering concepts should be considered at all relevant abstraction levels along the design trajectory, in order to enable the expression of designs at these abstraction levels. This raises the question: what are these relevant abstraction levels? At the same time the concepts at different abstraction levels should be properly related, which is necessary to perform design steps.

3.1.2 Correct Abstraction

Evidently, when an engineering concept is identified, its abstract representation in the formal model should be correct. This means that engineering details which are irrelevant to the considered abstraction level should be omitted. *But at the same time abstract notions which are im-*

1. It is quite strange that there is not a Protocol Conventions document.

proper to the engineering concept should not be introduced. Usually there is a kind of interaction, or bootstrap process, between identifying the relevant engineering concept and its abstract representation, leading to clarification at both sides.

The requirement not to introduce improper abstract notions is often violated. An example of abstraction which appears quite debatable is the notion of the eventual execution of an interaction in LOTOS once all involved processes are willing to participate in this execution. This notion was chosen in relation to the absence of a notion of time in LOTOS. But one may wonder whether this abstraction of time is consistent with engineering requirements. Things get worse when this notion is used to justify the implementation of a choice construct by one of its choice alternatives on basis of the reduction relation ([1]). It seems here that mathematics is forcing engineering and not vice-versa. In the same way, the atomicity of interactions (often interpreted as no time duration), arbitrary interleaving and exit impose quite nasty engineering problems.

The tendency to introduce improper abstractions seems to be often inspired by the fixation of many formalists on the criterion of correctness of specifications. This fixation might easily favour simplifications in the choice of the formal model, such as the possibility to define a single global state, which facilitate the verification of specifications against criteria such as deadlock, liveness and lifelock. Without diminishing the correctness requirement, it should never go at the cost of the usefulness of the formal model for pragmatic engineering purposes. Moreover, one should realize that many products which are incorrect in the sense that they still contain errors, appear quite useful, and actually are used at a large scale, in practice, even by formalists.

Inappropriate abstractions suppress the creativity of the designer as they force him to use tricks to express technical constructs that do not have a straightforward representation in the formal model. They lead to unwanted and tricky specifications and implementations.

Therefore formalists should seek cooperation with engineers and together carefully consider and agree on the engineering concepts and their abstraction rather than simply informing the engineers that a particular choice of abstraction adapts to engineering intuition.

3.1.3 Complete Set

The set of design concepts should be parsimonious. This implies that slightly diverging concepts should be avoided and replaced by one generalized, possibly parameterized, concept, or replaced by compositions of more elementary concepts. The set of engineering concepts and their abstractions should observe qualitative architectural principles such as generality, orthogonality, propriety, economy, etc. While observing these criteria, however, the set of concepts should be complete in the sense that all essential engineering needs should be covered. This implies that the parsimony criterion should not be taken into the extreme such that it becomes an excuse to limit the formal model while neglecting the needs of pragmatic engineering. The latter might greatly facilitate the development of a simple and complete formal semantics, but it forces at the same time the engineer to apply tricks, to combine formal and informal techniques, or, more likely, to abandon the formal model. If in a motorcar the breaks are eliminated because they cannot be integrated in the engine, the car will be of no practical use, notwithstanding the (formal) argument that the friction in the engine will *eventually* stop the car.

Given the possibility to choose, an engineer would prefer a language with a rich set of concepts, which can be selected in a systematic way, over one with a limited set. Our natural languages are also rich and can be handled, of course with various qualities, by almost everyone. Examples of concepts which are not supported by LOTOS are real concurrency, timing constraints, probabilistic constraints and internal actions.

3.1.4 Example

The requirements formulated so far may look straightforward. Nevertheless most of these requirements are poorly supported by currently available formal methods. We base our example on LOTOS and on an ad hoc notation for expressing causality between (inter)actions. The latter notation is introduced here since it supports the considered requirements in an intuitively appealing way.

Consider a simple system that performs the transfer of a word of 16 bits from one point to another. Figure 2 presents an abstract view of this system according to an OSI structure of service provider and service users. This figure also depicts the causal relationship between the request for data transfer (*req*) and the corresponding indication (*ind*).

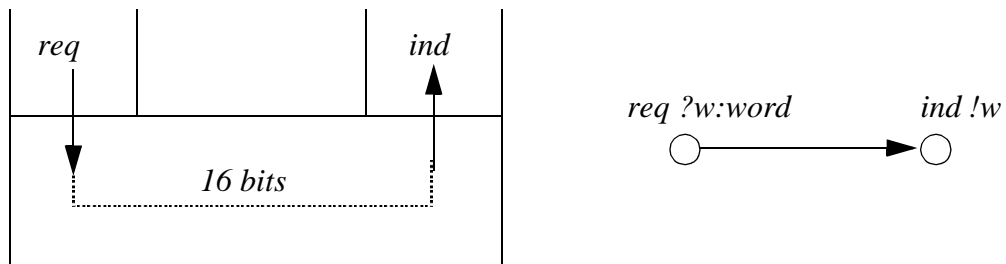


Figure 2: Simple Word Transfer Service Provider

This behaviour can be represented as follows:

$$S[req, ind] := req ?w:word ; ind !w ; exit \quad S = \{start \rightarrow req(w1:word), req(w1) \rightarrow ind(w2=w1)\}$$

Suppose that we want to refine this behaviour by introducing an internal action *p*, as is shown in Figure 3. The occurrence of *p* signifies the establishment of the same 16 bits at some location internal to the system. Therefore, *req* must occur before *p*, and *p* must occur before *ind*.

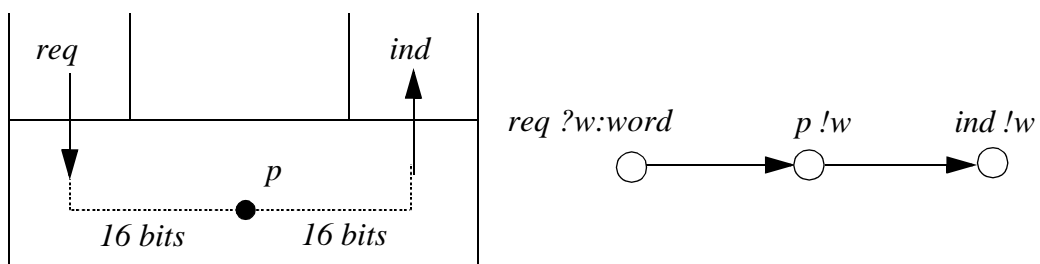


Figure 3: Introducing internal action *p*

The representation of internal action *p* in LOTOS can be done by making it hidden from the environment of the system being specified. It is assumed that in our ad hoc notation internal behaviour is intentionally not hidden:

$$S'[req, ind] := \text{hide } p \text{ in } req ?w:word ; p !w ; ind !w ; exit \quad S' = \{start \rightarrow req(w_1:word), req(w_1) \rightarrow p(w_2=w_1), p(w_2) \rightarrow ind(w_3=w_2)\}$$

The LOTOS expression may be considered as an improper use of the language, since LOTOS is based on interactions and we have used an interaction offer to represent an internal action. An alternative would be to define a structure of processes, in which two processes share an interaction corresponding to p . From a design point of view this may be considered over-specification because one may be not interested in defining such a specific structure yet.

Suppose now that we decompose the internal action p , such that the establishment of the original 16 bits in p is achieved by two actions p_1 and p_2 , each one responsible for one octet. We want at this point that p_1 and p_2 are independent of each other. Later design decisions may make these actions interleaved, or may prescribe a specific order. Figure 4 presents the resulting behaviour.

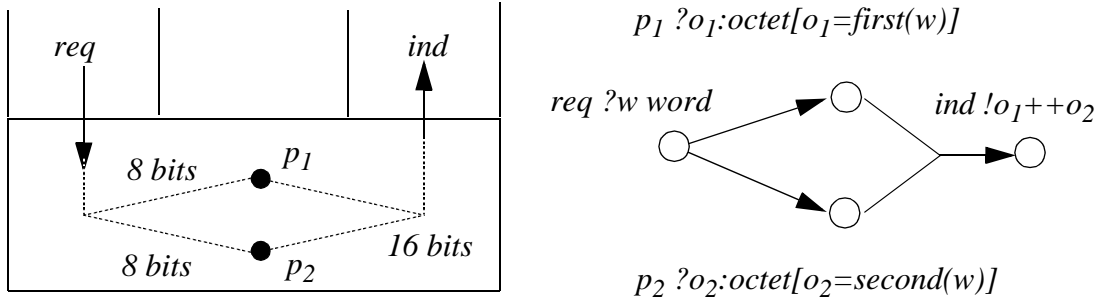


Figure 4: Decomposing internal action p

There is a causality relation between req and each of the internal actions, since the octet values of these internal actions depend on the 16 bits data of req . The 16 bits data of ind are generated from the octets of both p_1 and p_2 . Therefore these internal actions must have occurred before ind can occur.

The representation of this behaviour in LOTOS is problematic. First, the same observations made above with respect to action p hold here. Second, there are some problems related to the LOTOS interleaving semantics. This may be rather cumbersome for some types of behaviours, and brings extra difficulties for the verification of specifications, since the relationship between the interleaving semantics model and the original desired behaviour may not be straightforward. This is particularly true for complex behaviours, for which formal methods in fact should be helpful.

The example above can be represented in LOTOS and in the ad hoc notation in the following way:

```

S'' [req, ind] := hide p1, p2 in
req ?w:word ;
( p1 ?o1:octet[o1=first(w)] ;
exit (o1, any octet)
||| ( p2 ?o2:octet[o2=second(w)] ;
exit (any octet, o2) )
>> accept o1, o2:octet in
ind !o1++o2 ; exit

```

```

S'' = {start → req(w1:word),
req(w1) → p1(o1=first(w1)),
req(w1) → p2(o2=second(w1)),
p1(o1) ∧ p2(o2) →
ind(w2=o1++o2) }

```

This means that p_1 and p_2 can only be represented in LOTOS as interleaved actions (interactions). However making these actions interleaved has been mentioned above as a specific pos-

sible design decision, which does not necessarily have to be taken. In relation to the earlier mentioned concepts the LOTOS specification seems quite ambiguous and unclear with respect to what the designer originally meant to express.

3.2 Design Methodology

In a development environment supported by FMs (FM environment), the formal model should form the basis for the development of a design methodology that covers all essential aspects of system design. This methodology should provide pragmatic guidance to the system designer to develop the system starting at a high level of abstraction with the user requirements and finishing at a low level of abstraction with the implementation specification. To be able to develop such a methodology the formal model should be capable of supporting system design at all relevant levels of abstraction and supporting design transformations between these levels. The following levels of abstraction are seen as at least necessary in a design methodology:

1. The definition of the system while embedded in the user environment. This design allows to express the user requirements at a high level of abstraction without being forced to decide how the responsibility for these requirements should be distributed over the system and the user(s).
2. The definition of the system in terms of a service provider. This requires that the requirements as defined in step 1 are distributed over the system and the user(s). It allows to express the design of the service provider only in terms of what functions are provided by the system, without being forced to distribute these functions over the parts that ultimately will constitute the service provider.
3. The definition of the service provider in terms of a set of cooperating parts. This requires that the functions of the service provider are decomposed and distributed over the parts that constitute the internal structure of the provider. It allows to define the functions of each individual part separately at a certain abstraction level without being forced to make more detailed implementation decisions for these functions.
4. The replacement of the abstract interfaces of the parts by real interfaces. This allows to select the appropriate interfaces as determined by engineering requirements.
5. The definition of the implementation structure that replaces the relationships between the real interfaces of each individual part. To achieve this, steps 3 and 4, but applied to a part, can be repeated iteratively and in various orders.

Reflecting these design structures in a standard way in the structure of design specifications leads to the notion of specification styles ([15]). The availability of specification styles may greatly facilitate the development of designs.

It appears that most FDTs are too restrictive to cover the whole design trajectory from high level abstract specification to low level implementation specification, making it extremely hard to develop a sufficiently comprehensive design methodology that is capable of bridging this large gap. Moreover it appears that there exists little insight, agreement, and common understanding of how such an FDT and comprehensive methodology should look like. This implies that, in addition to the necessary high investments, such methodologies can only be developed for very few FDTs. Yet the availability of a comprehensive design methodology is essential for industrial acceptance of FMs.

3.3 Design Support Tools

Tools form in any technical discipline an important aid to the engineer and directly aim at improving productivity. The same should apply to the field of engineering (distributed) information systems and to a FMs environment that aims at supporting this field. To be effective, tools should support all aspects of a comprehensive design methodology as discussed in Section 3.2 above. The feasibility of such tools is largely determined by the appropriateness of the formal model and the availability of such a comprehensive design methodology. In a FMs environment the individual tools should not be self-standing but form a component of a consistent toolkit architecture. By this we mean that the tools should focus on different and clearly delimited aspects of the design trajectory, should be based on a consistent internal representation which makes it possible to easily switch back and forward between tools, and should have a consistent interface with respect to each other and to the user. The following categories of tools are seen as indispensable components of a comprehensive toolkit: *design specification support tools* such as syntax and semantics checkers and specification structuring tools, *design analysis support tools* such as simulators and verifiers, *design transformation support tools* such as transformers and compilers, and *conformance checking tools* such as test tools. The Lotosphere project has done excellent pioneering work in the development of such a toolkit ([12]).

To achieve industrial acceptance, however, there are additional requirements that should be carefully observed:

1. Tools should be easy to learn and use! It is extremely discouraging if engineers have to study a long time on a tool before they can use it and then time and again find out that it has all kinds of unexpected, incomplete, and inconsistent behaviours. A well designed toolkit architecture and a consistent and attractive user interface supports this requirement.
2. Tools (and FMs in general) should be applicable to large and complex designs. Tools that are demonstrated for toy problems, and indeed in practice appear to be limited to toy problems, are of no use and undermine the credibility of FMs.
3. Tools should be produced, documented and maintained at an industrial level. This requirement often demonstrates the weakness of the FMs community to comply with industrial needs.

The above requirements are often violated. Many formalists appear to be only interested in sophisticated tools that embody some interesting theory without taking the trouble to invest in less interesting but pragmatically highly relevant matters. The latter is often considered as the dirty work that should be left over to some commercially interested party. This self-centered view, however, undermines the cause of FMs. A drudgery at the short term appears often a strategic investment to safeguard the more elegant work at the longer term.

Given the requirements for a comprehensive toolkit as discussed above, and the very high investments necessary to develop it, it appears a naive thought that such a toolkit can be developed for any formal model and design methodology. Therefore it seems expedient to ascertain first the design methodology and its coverage before making such investments. On the other hand, once these pre-requisites are available, their completion with a comprehensive toolkit will appear indispensable for an FMs breakthrough in industrial acceptance.

3.4 Migration Path

New methods should be useful next to existing, more conventional, methods and not have the nature of being conflicting with them and pushing them out ([16]). It is quite naive to assume that FMs will be introduced overnight in industry. Engineers simply will not put their produc-

tivity at risk. Rather a FM should prove its capability and a carefully considered migration path should be traced out that allows its step by step introduction. This allows to gradually build up confidence, which is rightly is not assumed beforehand by the engineer, in its usefulness and trustworthiness.

The possibility of a step by step introduction puts requirements to the way the FM is structured. This structuring should be such that a simple subset of the FM can be used to tackle the less demanding engineering problems, whereas more sophisticated methods should only be introduced when the complexity of the engineering problem increases. A FM that is only useful when it is applied in full, and that requires a lot of theoretical study, will in practice not be applied at all.

3.5 Limited Number of Formal Methods

The logical consequence of the above reasoning, and probably the most frustrating one for the FMs community, is that FMs should only be developed for a very limited number of formal models. There are at least two reasons for this: First it is practically impossible for the industry to assess the unwieldy amount of formal models and associated FMs and select the useful ones. Second, as mentioned above, it is practically only possible to make the necessary high investments for a full blown FM, including the design methodology and the tools, for a limited number of formal models.

This implies that, as long as the FMs community exhausts itself in developing all kinds of diverging theories, while taking little account of the actual engineering needs, they will never have the chance of developing full blown FMs that will be accepted by industries. Instead, the status quo of low credibility will be maintained that way.

To break through this situation (part of) the FMs community should come together and *get organized*. The objective should be to identify and select a limited number of distinct but really pragmatic formal models and join efforts to develop full blown FMs for each one of them while avoiding all kinds of slightly diverging, yet incompatible, developments. In this endeavor sophistication should be subordinate to appropriateness and usefulness. This endeavor may also require to do, like in tool developments, some “dirty” work like moving the politics in a standardization organization.

3.6 Accessibility

In order to be used, FMs should be accessible, i.e. they should be well documented and good course and tutorial materials should be made available for educating engineers. If not accessible and understood, FMs will not be used. It needs no arguing that the more appropriate the formal model, the easier the engineer will digest the educational material. Courses and tutorials are needed at least at introductory level, at advanced level, and at management level. This material should explain the formal model, the design methods, and the design support tools. It should also discuss the relative benefits of the FMs and illustrate these benefits on basis of a range of small to more complex examples and exercises.

Evident as it seems, it is astonishing though to observe how little tutorial material and industrial level tool documentation is available for FMs. One of the few examples of tutorial material on FDTs is [11]. However most LOTOS based FMs do not have industrially utilizable tutorials in spite of the large investments made in their development. Here again it seems that writing the tutorials and the documentation is considered as the dirty, less rewarding, work at the cost of the longer term repercussion of a lower than possible dissemination and acceptance.

Moreover, formalists seem to be only interested to only talk to formalists in the mathematical language of the formalists. They do not bother to present their results in a style that makes them accessible for the engineer. If the latter is studying hard on a paper, and after a week of digesting mathematics finds out that an abstraction is made which makes the contents of the paper useless for his purpose, he will not be tempted to study another paper with the same enthusiasm.

3.7 Intermediate Conclusion

Pursuing the traditional criteria for developing FMs, such as formal syntax and semantics, correctness, unambiguity and conciseness, has in the recent past led to quite sophisticated and admirable FM results for which the FMs community deserves a compliment. Yet it led to a marginal uptake in industry. To achieve this uptake the FMs community should seriously observe the pragmatic requirements of the application area, the engineering of (distributed) information systems mentioned in this section. If such requirements are ignored the chances of getting further isolated from the engineering community, at the cost of jeopardizing the cause of FMs and reducing research funds, seem quite realistic.

It occurs to us that many of the pragmatic engineering requirements directly or indirectly relate to the appropriateness criterion, i.e. the availability of a complete set of correct abstractions of relevant engineering concepts. For these reasons more research should be carried out in the area of design methods and the basic design concepts that support these methods as shown in Figure 1. The objectives of this research should be: to obtain better insights in the engineering needs of the area of distributed information systems design, to develop the basic design concepts that determine a comprehensive design methodology for this area, and to develop an appropriate formal model that underlies a comprehensive FMs engineering environment. These insights should help existing FMs to improve in the direction of a better industrial applicability.

4 Design Structuring

Often a system specification is interpreted (i) as to constrain the behaviour of the system's environment, or (ii) such as it does not state what happens in unspecified cases ([10]). In other words a system specification defines what the system does under pre-defined conditions which should be known and respected by the system's environment, and says nothing about the unknown pieces of behaviour. Based on these interpretations, a system is often defined as an entity *separately* from its environment. This can be done in terms of the possible orderings of *interactions* between the system and its environment, the information values established in these interactions, and the constraints on these values, *all as imposed by the system*. The system's environment can be defined in the same way.

The definitions of the system and its environment *together* determine a *common behaviour*: *what* interactions in *what* order and with *what* value attributes can actually be established. This common behaviour is what the user is really interested in, and therefore it should be specified *first* and be used later to derive the behaviour of the system. We call this common behaviour the *interaction system* between the system and its environment. In the definition of the interaction system only the *result* of each interaction is defined while abstracting from the many different ways in which the system and its environment may contribute to these results. Therefore interaction systems are concerned with *integrated interactions*, which we henceforth call *actions*.

The interaction system defines, at a higher level of abstraction than the definition of a system, *what* may happen in terms of possible actions, not *how* it may happen in terms of possible in-

teractions. This means that constraints can be freely placed on the way the system and its environment participate in interactions as long as the desired actions are implemented, allowing design freedom for choosing the definition of interactions. Although all integrated interactions are actions, some actions cannot be called integrated interactions, since in the course of the design process one may choose not to decompose certain actions into interactions and consequently not to distribute them over parts.

Our design methodology thus starts with the definition of an interaction system requiring as a basic design concept the notion of action. In subsequent design steps an interaction system is decomposed into parts (e.g. a system and its environment). Actions may be decomposed into interactions, which are allocated to and distributed over these cooperating parts. In order to define the interaction system we first develop a model for monolithic behaviour definition and causality-oriented behaviour composition.

Experience has shown that complex systems can often better be specified in a structured way in order to be conceived, understood, manipulated and maintained. In particular the constraint-oriented specification style ([15]), where a behaviour is structured as a composition of sub-behaviours acting as constraints, has been proposed. This way of defining an interaction system requires that certain actions are decomposed into interactions, which are assigned to constraints. This design step can actually be used to *prepare* the decomposition of a functional entity (e.g. an interaction system) into parts (e.g. a system and its environment), since the constraints can be assigned *in various ways* to these parts.

Our design methodology thus continues with decomposing the interaction system into a set of constraints, requiring as basic design concepts the decomposition of an action into interactions as discussed above and the notion of constraints. In this design step the above mentioned design freedom can be exploited. The next design step in our design methodology is the allocation of the constraints to the system and its environment. This step may sometimes require to repeat the previous design step before the allocation can take place.

We illustrate these design structuring principles with a simple example illustrated in Figure 5, viz. a question-answer service. The question-answer service specifies that the establishment of a question at the calling side (q_req at Q) is followed by the establishment of the same question at the called side (q_ind at A), which is then followed by the establishment of an answer at the called side (a_req at A) and the establishment of the same answer at the calling side (a_ind at Q). Figure 5 shows the first three design steps in the development of the question-answer service, relating them to the first three levels of abstraction discussed above. In this figure we illustrate the design choice of assigning the relationships between q_req and q_ind , and a_req and a_ind to the system, i.e. the service provider, and of assigning the relationship between q_ind and a_req to the environment, i.e. the service users.

The next global steps in our design methodology are the decomposition of the Service Provider into system parts and the replacement of the abstract interfaces by the real interfaces, completing the abstraction levels as discussed in Section 3.2. For brevity reasons we cannot show these other two steps.

5 Basic Architectural Concepts

The concepts which allow the different forms of design structuring introduced in Section 4 are elaborated in this section. Section 5.1 presents a model for monolithic behaviour definition,

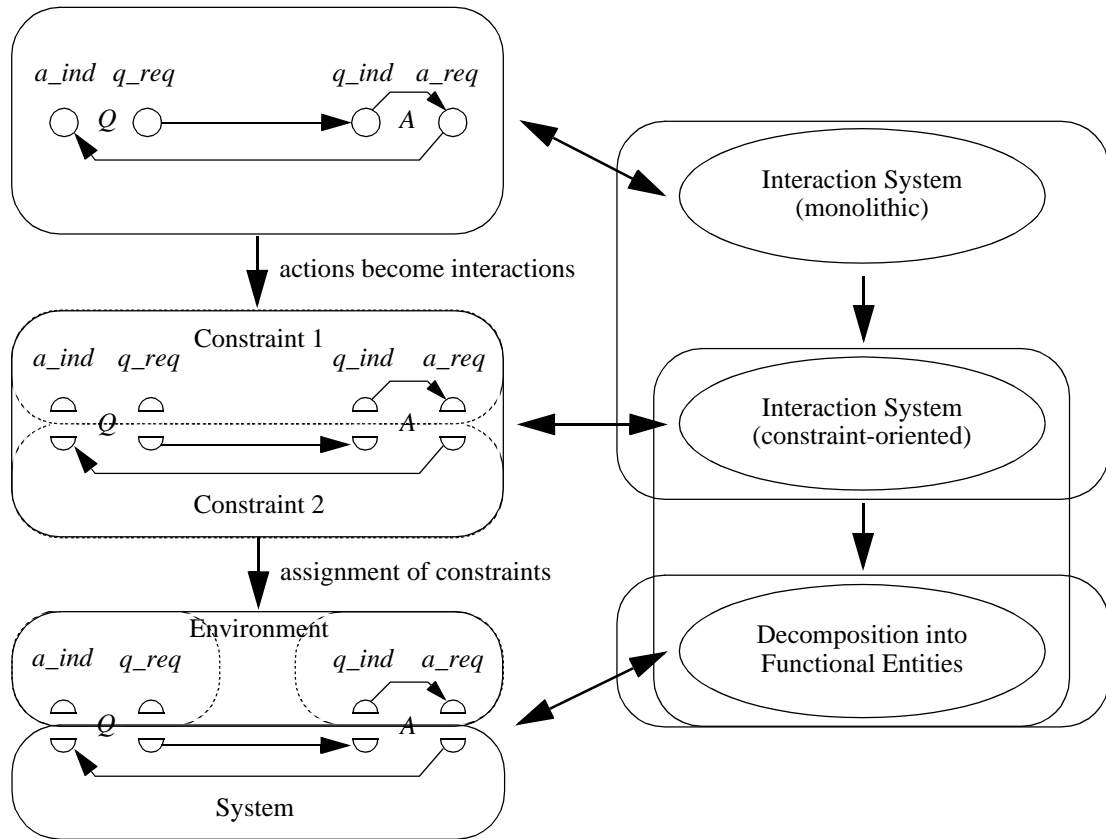


Figure 5: First Three Steps of the Design Trajectory

Section 5.2 extends this model to allow causality-oriented behaviour composition and Section 5.3 introduces constraint-oriented behaviour composition.

5.1 Monolithic Behaviour Definition

Actions, interactions, and behaviour, represented in terms of causality relations between actions and interactions form the basic concepts that allow the definition of behaviour.

For brevity reasons we mention in the sequel only actions where statements also apply to interactions. If actions and interactions need to be distinguished, it is explicitly mentioned unless it follows clearly from the context. In the figures actions are drawn as circles and interactions as circle-segments¹.

5.1.1 Attributes of Actions and Interactions

We identify, based on our design experience, the following attributes of actions:

- *Location attribute*: defines the location at which the action occurs;
- *Time attribute*: defines the moment or period of time when an action occurs or can occur;
- *Values of information*: defines the values of information established in the action;
- *Functionality attribute*: defines the set of values of information passed to this action by previous actions and that may be referred to by successive actions;

1. Actually we have already done this in previous figures.

- *Probability attribute*: defines the probability that an action occurs according to its definition, once this action is *enabled*.

These attributes fully characterize actions and can be used in the definition of their relationships in order to define the behaviour of functional entities. Attributes of actions may make references to attributes of previously occurred actions with which there exists an enabling causal relationship.

Action attributes and their relationships are also important for the definition of implementation notions. In action decomposition, for example, all attributes may be decomposed.

We assume that we can always distinguish, and thus uniquely identify, all actions in behaviour definitions. In the examples below we use unique *action identifiers* to indicate their occurrence or non-occurrence.

5.1.2 Behaviour Definitions

We define the behaviour of an interaction system, a system's environment, a system, and system parts, by a set of causal relationships amongst actions and interactions. Behaviour definitions contain the following elements:

- *Initial action*: an action which occurs independently of the occurrence of other actions of the defined behaviour, possibly with an initial set of attributes. It may occur spontaneously when the behaviour is instantiated (*start* action), or it may be enabled by other behaviours to which the behaviour is linked (*entry points*);
- *Causality context*: the causality context of an action defines the role of this action in a behaviour;
- *Exit condition*: a behaviour is said to *exit* if it enables initial actions of another behaviour. This enabled behaviour is then allowed to start, allowing to make references to the attributes of the exit condition.
- *Termination condition*: an action is said to terminate a behaviour if no more actions or other behaviours are enabled by it, characterizing *deadlock*.

The conditions for the occurrence of an action of a behaviour B are defined in a *causality relation* between the other actions of B and this action. We say that a causality relation defines the conditions which enable and constrain the occurrence of an action. The reason for using causality as the basis for defining behaviour is based on the fact that a distributed behaviour generally does not have a single global state, but rather a changing set of 'sub-states'. These 'sub-states' can be derived from the causality relations.

5.1.3 Basic Causality Relations

We define the causality relation $a_1 \rightarrow a_2$ as:

the occurrence of action a_1 is a condition for the occurrence of action a_2 ; attributes of a_2 may refer to attributes of a_1

The causality relation above says nothing about the conditions for the occurrence of a_1 . These are defined in the causality relation of a_1 , which is not part of the causality relation of a_2 . This implies that the possible occurrence of a_2 can only be determined after evaluation of all causality relations that lead to the occurrence of a_1 .

The occurrence of an action may also depend on the non-occurrence of another action, characterizing a type of causality, called *conflict*.

We define the causality relation $\neg a_1 \rightarrow a_2$ as:

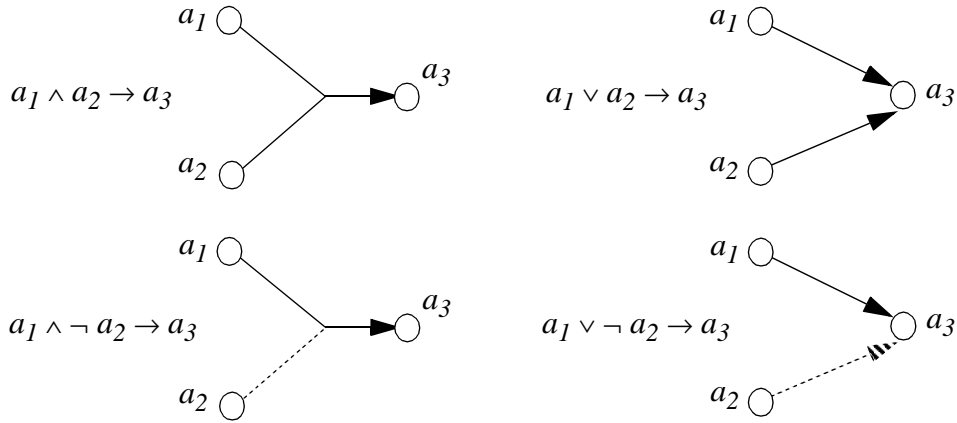


Figure 7: Some Causality Relations and Their Graphical Notation

5.1.5 Conditions and Constraints on Attributes

Action attributes play distinct roles in a causality relation, depending on whether an action is a condition or a resulting action:

- *Attributes in conditions*: can be used to define specific premisses on values of action attributes in a condition for the resulting action to occur;
- *Attributes in resulting actions*: are used to define the allowed values of the attributes of the resulting action.

Example

$$a_1 (v_1: \text{Data}) [v_1 < 10] \rightarrow a_2 (v_2: \text{Data}) [v_2 = v_1 + 3]$$

This causality relation states that only in case a_1 happens with a value v_1 smaller than 10, a_2 is allowed to happen. If a_2 happens its value will be $v_1 + 3$. In case a_1 does not happen or happens with v_1 greater or equal to 10, the condition for a_2 is false, and a_2 is not allowed to happen.

Similar discussions also apply to the other action attributes. Due to size limitations we do not further elaborate on this point in this paper.

5.1.6 Finite Monolithic Behaviour

A finite behaviour can be represented by a set of causality relations, one relation per action of this behaviour. Consider the following example:

$$B := \{ \begin{array}{l} \text{start} \rightarrow a_0, \\ \text{start} \wedge \neg a_0 \rightarrow a_1, \\ a_1 \vee a_0 \rightarrow a_2, \\ a_1 \wedge \neg a_2 \rightarrow a_3, \\ a_2 \rightarrow a_4 \end{array} \}$$

$\text{start} \rightarrow a_0$ states that a_0 is enabled from the beginning of the behaviour, i.e. it does not have to wait for other actions. $\text{start} \wedge \neg a_0 \rightarrow a_1$ implies that while a_0 does not happen a_1 is allowed to happen, therefore at the beginning of the behaviour a_1 is also enabled. This means that both a_0 and a_1 are initial actions of B . The behaviour definition B also determines the causality context of all its actions. For example, a_2 is the resulting action in the causality relation $a_1 \rightarrow a_2$,

and appears in the causality relations $\{a_1 \wedge \neg a_2 \rightarrow a_3, a_2 \rightarrow a_4\}$. This completely defines the role of a_2 in B .

5.2 Causality-oriented Behaviour Composition

Causality relations, as presented so far, are a compact and parsimonious notation to define relationships between actions, but lack structuring. Furthermore this notation is restricted to finite behaviours. Therefore this section builds on the previous section, presenting some mechanisms to structure designs and to represent repetitive and infinite behaviour.

5.2.1 Causal Composition with Entry and Exit

Causal compositions of behaviours are characterized by the fact that conditions inside an instance of behaviour enable actions of another instance of behaviour, in a similar way as conditions on actions enable result actions in causality relations.

Entries and exits are introduced in our design model as mechanisms to represent causality-oriented composition of behaviours. Suppose B_1 and B_2 are behaviours, defined as sets of causality relations, and that B_1 has one exit and B_2 has one entry. A causality-oriented composition between B_1 and B_2 can be defined by combining the conditions of the exit of B_1 and the action(s) of the entry of B_2 , such that the conditions of the exit of B_1 become conditions to the action(s) of the entry of B_2 . This can be generalized to more than one entry, more than one exit, or both.

We illustrate this with the following example:

$$B_1 := \{ \text{start} \rightarrow a_1, \text{start} \rightarrow a_2, \\ a_1 \wedge a_2 \rightarrow a_3, \\ a_3 \rightarrow a_4, \\ a_3 \rightarrow a_5, \\ a_4 \wedge a_5 \rightarrow \text{exit} \}$$

$$B_2 := \{ \text{entry} \rightarrow a_6, \text{entry} \rightarrow a_7, \\ a_6 \wedge a_7 \rightarrow a_8 \}$$

The statements $\text{entry} \rightarrow a_6, \text{entry} \rightarrow a_7$ mean that a_6 and a_7 can be enabled by coupling a condition to this *entry*. $a_4 \wedge a_5 \rightarrow \text{exit}$ means that if a_4 and a_5 happen, the exit condition is true. We can define a sequential composition between B_1 and B_2 in the following way:

$$B := \{ B_1(\text{exit}) \rightarrow B_2(\text{entry}) \}$$

This means that B_1 is coupled to B_2 such that the conditions of the *exit* of B_1 become the conditions for the actions of the *entry* of B_2 . The resulting behaviour can be obtained by the ‘short-circuit’ of the *exit* of B_1 and the actions of the *entry* of B_2 .

Figure 8 depicts this example, showing that the conditions of the exit of B_1 have turned into conditions for the occurrence of a_6 and for the occurrence of a_7 .

Exits can also be used to refer to negation of conditions. Consider the following example:

$$B_1 := \{ \text{start} \rightarrow a_1, \text{start} \rightarrow a_2, a_1 \vee a_2 \rightarrow \text{exit} \}$$

$$B_2 := \{ \text{entry} \rightarrow a_3 \}$$

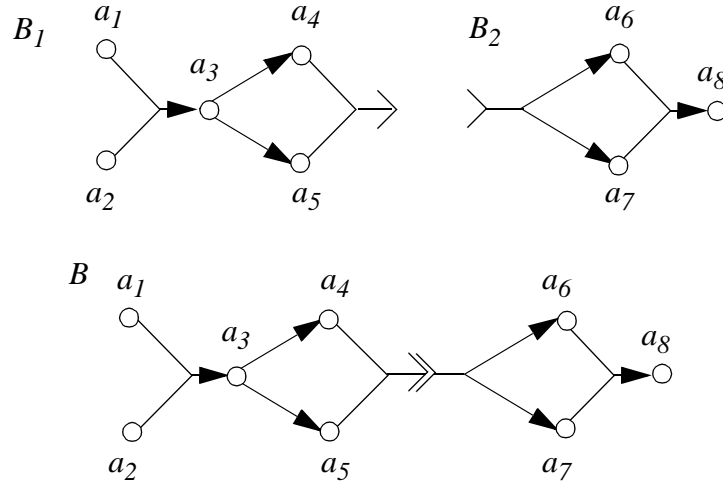


Figure 8: Example of Causality-Oriented Behaviour Composition

The behaviour definition $\neg B_1(\text{exit}) \rightarrow B_2(\text{entry})$ makes the negation of the exit of B_1 , i.e the non-occurrence of both actions a_1 and a_2 , a condition for the occurrence of a_3 .

5.2.2 Multiple Entries and Exits

We generalize the exit/entry constructs, by considering that a behaviour may have more than one exit conditions or entry points or both. The consequence is that, in order to allow a proper combination of exit/entry pairs, exits and entries must be identified.

We illustrate this with an example:

$$B_1 := \{ \text{start} \rightarrow a_4, \text{start} \rightarrow a_5, \text{start} \rightarrow a_7, \\ a_4 \wedge a_5 \rightarrow \text{exit}_1 \\ a_7 \rightarrow \text{exit}_2 \}$$

$$B_2 := \{ \text{entry}_1 \rightarrow a_8, \\ \text{entry}_2 \rightarrow a_9 \}$$

The behaviour definition $\{B_1(\text{exit}_1) \rightarrow B_2(\text{entry}_1), B_1(\text{exit}_2) \rightarrow B_2(\text{entry}_2)\}$ means that exit_1 of B_1 is connected to entry_1 of B_2 , and that exit_2 of B_1 is connected to entry_2 of B_2 . It is important to remark that $B_1(\text{exit}_1)$, $B_1(\text{exit}_2)$ and $B_2(\text{entry}_1)$, $B_2(\text{entry}_2)$ refer to the same instance of B_1 and B_2 , respectively.

Figure 9 illustrates the effect of this combination mechanism with generic behaviours B_1 , B_2 , B_3 and B_4 . Notice that the exit/entry constructs define a line that delimits the behaviours by decomposing the causality relations. This mechanism allows us to structure a monolithic behaviour in sub-behaviours, in such a way that compositions of behaviour definitions can be created.

5.3 Constraint-oriented Behaviour Composition

An important structuring technique identified in [15] for the representation of behaviours is the *constraint-oriented style*. According to this style, a behaviour is represented as a conjunction of constraints on actions, which are described in separate processes.

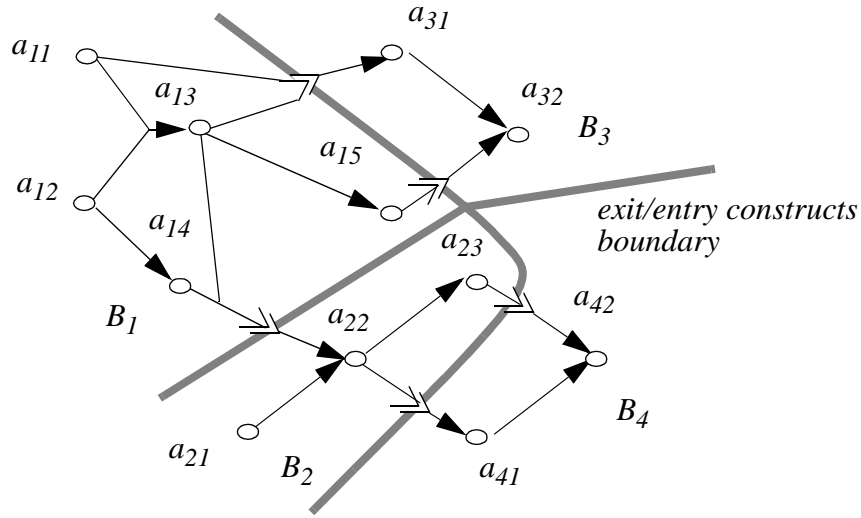


Figure 9: Generalized Exit/Entry Constructs

The consideration of this approach in our design model forces us to represent some actions in a distributed form. This happens since the global causality relation of each action to be distributed among multiple constraints is defined as a collection of causality relations in the different behaviours that represent these constraints. The combination of this collection of causality relations determine the desired global causality relation for each distributed action.

After the global causality relation of the distributed actions has been defined in separate behaviours, these behaviours can be assigned to the system and to its environment. Actually only at this point an action is transformed to an interaction.

We consider as an example a behaviour B , which contains a causality relation $\mathbf{a} \wedge \mathbf{b} \rightarrow \mathbf{c}$. We also suppose that \mathbf{a} , \mathbf{b} and \mathbf{c} may be or may be not distributed over behaviours B_1 and B_2 . Actions which are not distributed are represented in this example with **bold**, while actions which are distributed are represented in *italics* when they turn into interactions. This analysis concentrates exclusively on the causality relation of \mathbf{c} without loss of generality.

We spare the reader from the boring exercise of considering all possibilities for the decomposition of \mathbf{a} , \mathbf{b} and \mathbf{c} , but rather present one of them below:

Distribute condition \mathbf{a} (or \mathbf{b}) and the result \mathbf{c}

The correct options for the decomposition of the causality relation $\mathbf{a} \wedge \mathbf{b} \rightarrow \mathbf{c}$ by distributing condition \mathbf{a} and result \mathbf{c} over B_1 and B_2 should preserve the original causality relation. These options are presented below and illustrated in Figure 10:

1. the whole causality relation is placed in one of the behaviours. In Figure 10 the original causality relation is placed in B_2 , while B_1 does not contribute to the causality relation. In Figure 10, $\ast \rightarrow \mathbf{c}$ indicates that \mathbf{c} must be enabled in B_1 at the moment that \mathbf{a} and \mathbf{b} occur. Since \mathbf{a} belongs to B_1 , \mathbf{c} may be enabled in B_1 from the beginning, either by an initial action, or by any other conditions which are also necessary conditions of \mathbf{a} .
2. the whole causality relation is placed in one of the behaviours, but part of the causality relation is duplicated in the other behaviour. In Figure 10 we have placed the whole original causality relation in B_2 and the causality between \mathbf{a} and \mathbf{c} is duplicated in B_1 . Although the duplication of constraints in B_1 and B_2 may seem technically undesirable, in the more general

case, where references to attributes are possible, this kind of decomposition may be used to express *separation of concerns*.

Suppose for example that in **a** two values of information v_1 and v_2 are established, and that both of them are used for the determination of the values of **c**. In this case we can use B_1 to constrain the dependencies with respect to v_1 and B_2 to constrain the dependencies with respect to v_2 . If v_1 and v_2 characterize two distinct aspects of the functions of behaviour B , this structuring in constraints shall be preserved in later phases of design and may be found back in the implementation of the system.

- the original causality relation is decomposed over both behaviours. In Figure 10 the causality between **a** and **c** is placed in B_1 and the causality between **b** and **c** is placed in B_2 . The combination of these constraint yield the original causality relation.

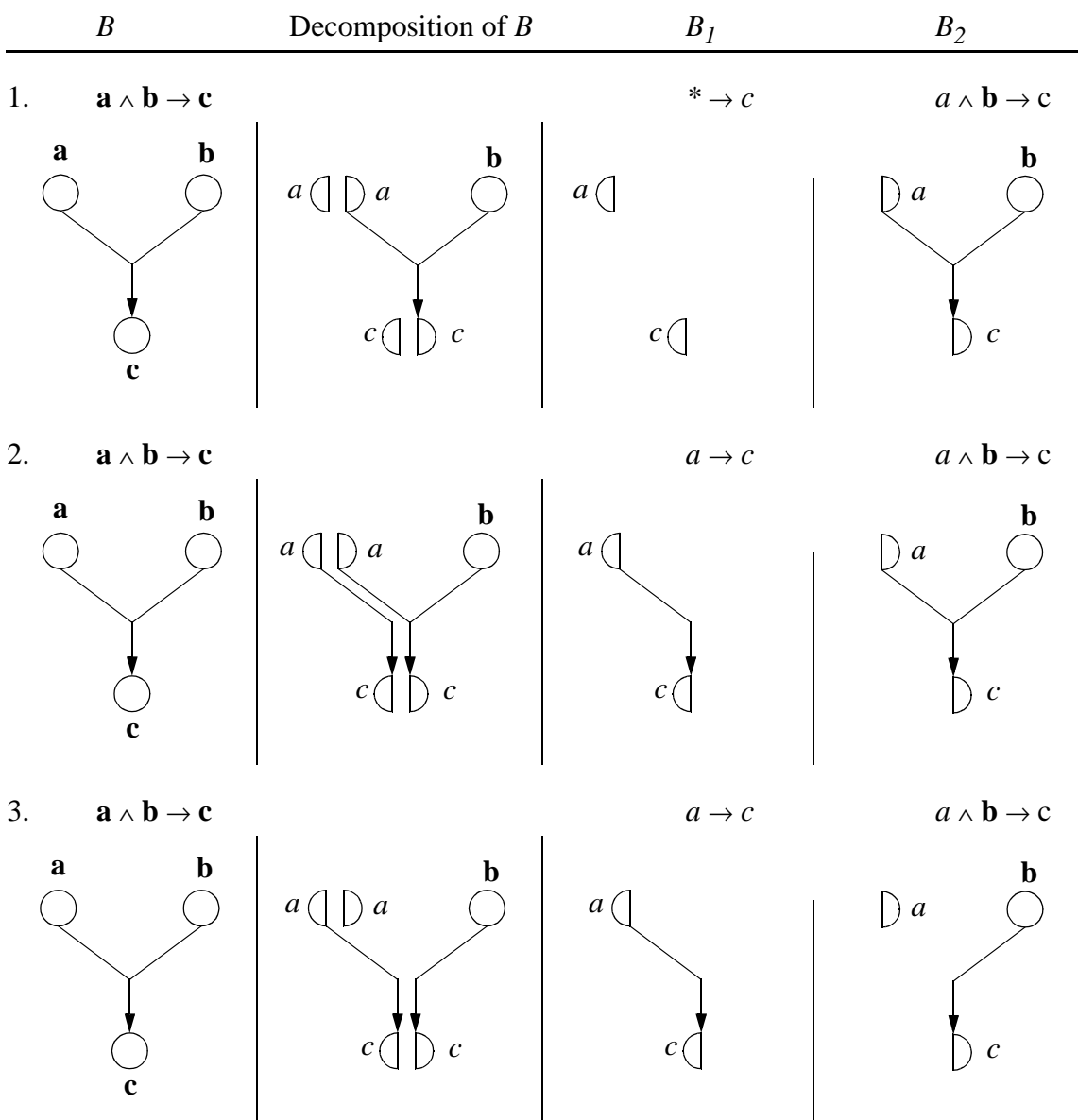


Figure 10: Three Possible Decompositions in Constraints

The mirror images in which B_1 and B_2 are exchanged, and the distribution of \mathbf{b} instead of \mathbf{a} are also considered here. Similar reasoning can be applied to other distributions of \mathbf{a} , \mathbf{b} and \mathbf{c} , and to causality relations of arbitrary complexity, involving disjunctions, conjunctions, occurrences and non-occurrences.

Figure 11 illustrates the effect of composition of constraints with four arbitrary behaviours B_1 , B_2 , B_3 and B_4 .

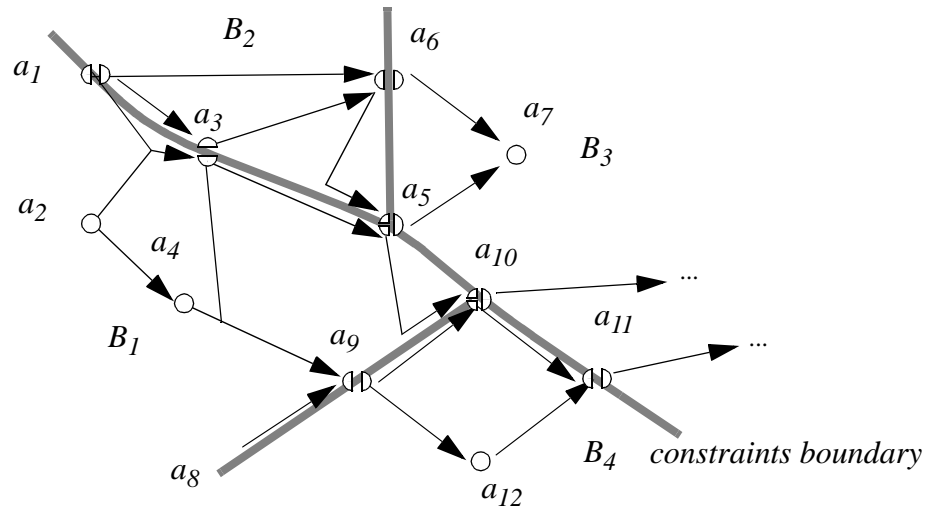


Figure 11: Arbitrary Composition of Constraints

6 Conclusions

The reluctance of the industry to introduce advanced FMs on a broad scale into their daily practice of designing and developing products should not simply be put on the account of their unwillingness or lack of understanding. Rather it should be put on the account of the FMs community of not being capable of turning FMs into real industrial tools. Current FMs are too self-willed and do not attend industrial requirements. They appear not broadly applicable, do not fit the purpose, frustrate the engineer, and are difficult to learn. Formal descriptions of designs appear incomprehensible, very costly to produce, and of little practical use to the average engineer.

If FMs are to become industrially applicable then the FMs community will have to start thinking, behaving, and working in an industrial way. This community will have to develop formal models that suit the needs of the application area and that are supported by comprehensive design methodologies and design support tools. To achieve these goals the FMs community needs to get organized, with the purpose to select only a limited set of formal models and to develop credible full blown FMs. In these FM developments proper attention should be given to provide ample and adequate educational material and to trace out proper migration paths to guide the step by step introduction of FMs in industry.

If these pragmatic requirements are not properly observed, the current reluctance of industry, may easily lead to what we may call an “FMs crisis”, a crisis of which the FMs community certainly will suffer most. Undoubtedly when the discipline of engineering distributed information systems gets mature it will use FMs as a matter of routine, like all other mature technical sciences do. However the question can be posed: will these methods be developed by formalists that understand the engineering needs, by engineers that understand formalisms, or by formalists and engineers working together?

Many of the engineering needs are directly related to the appropriateness criterion, the provision of a complete set of correct abstractions of relevant engineering concepts. It is suggested that the FMs community pays more attention to the study of basic design concepts and the design methods that are based on them in order to develop the material on which an appropriate formal model should be based. This paper presents some initial results of recent research that aims at getting a better insight in this area.

References

- [1] E. Brinksma, G. Scollo, and C. Steenbergen. LOTOS specifications, their implementations and their tests. In G. von Bochmann and B. Sarikaya, editors, *Protocol Specification, Testing and Verification VI*, pages 349–360, Amsterdam, 1987. North-Holland.
- [2] M. Gamble and C. R. Taylor. The CCSDS protocol validation programme inter agency testing using LOTOS. In J. Quemada, J. M. nas, and E. Vazquez, editors, *Formal Description Techniques, III*, pages 319–326, Netherlands, 1991. Elsevier Science Publishers B.V. (North-Holland).
- [3] ISO. Open Systems Interconnection, Reference Model, Version 4, June 1979. ISO/TC 97/SC 16/N277.
- [4] ISO. LOTOS description of the Session Protocol, 1989. ISO/IEC TR9572.
- [5] ISO. LOTOS description of the Session Service, 1989. ISO/IEC TR9571.
- [6] ISO. Formal description of ISO 8072 in LOTOS, 1992. ISO/IEC TR10023.
- [7] ISO. Formal description of ISO 8073 (Classes 0, 1, 2, 3) in LOTOS, 1992. ISO/IEC TR10024.
- [8] ISO/IEC JTC1/SC21. Recommendations on the use of Formal Description Techniques, Apr. 1993. ISO/IEC JTC1/SC21 N 7761.
- [9] P. W. King. Formalization of protocol engineering concepts. *IEEE Transactions on Computers*, 40(4):387–403, April 1991.
- [10] L. Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, January 1989.
- [11] K. J. Turner, editor. *Using Formal Description Techniques*. John Wiley & Sons, Great Britain, 1993.
- [12] P. van Eijk. Tool demonstration: The Lotosphere integrated tool environment Lite. In K. Parker and G. Rose, editors, *Formal Description Techniques, IV*. North-Holland, 1992.
- [13] C. A. Vissers, L. Ferreira Pires, and J. van de Lagemaat. Lotosphere, an attempt towards a design culture. In T. Bolognesi, E. Brinksma, and C. A. Vissers, editors, *Third Lotosphere Workshop and Seminar, Workshop Proceedings*, volume 1, pages 1–30, 1992.
- [14] C. A. Vissers and G. Scollo. Formal specification of OSI. In G. Muller and R. Blanc, editors, *International Seminar on Networking in Open Systems*, volume 248 of *Lecture Notes in Computer Science*, pages 338–359. Springer-Verlag, 1987.
- [15] C. A. Vissers, G. Scollo, M. van Sinderen, and E. Brinksma. Specification styles in distributed systems design and verification. *Theoretical Computer Science*, 89:179–206, 1991.
- [16] D. Weber-Wulff. Selling formal methods to industry. In J. Woodcock and P. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*, pages 671–678, Berlin, 1993. Springer-Verlag.