The Usage of Stochastic Processes in Embedded System Specifications

Axel Jantsch, Ingo Sander, Wenbiao Wu
Royal Institute of Technology, Stockholm, Sweden

ABSTRACT

We review the use of nondeterminism and identify two different purposes. The *descriptive purpose* handles uncertainties in the behaviour of existing entities. The *constraining purpose* is used in specifications to constrain implementations. For the specification of embedded systems we suggest a *stochastic process* σ instead of nondeterminism. It serves mostly the descriptive purpose but can also be used to constrain the system. We carefully distinguish different interpretations of these concepts by the different design activities simulation, synthesis and verification.

1. INTRODUCTION

Nondeterminism as a modelling concept has been used with two different objectives. One objective is to capture an aspect of the world which is not completely known and which behaves in an unpredictable manner. We call this usage the *descriptive purpose*. The second objective is to designate different possibilities for implementation, which we call the *constraining purpose*.

Following Dennis and Gao [4] we use the terms "determinate" and "deterministic" in the following sense. An entity is *deterministic* if its entire internal mechanism is fully functional, i.e. in each part and in each step the same output is produced for the same input. An entity is *determinate* if its externally visible behaviour is functional, i.e. if the entity always produces the same output for the same input. Thus, a nondeterministic system may be determinate but a nondeterminate system cannot be deterministic. Our interest here is mainly in what can be observed from outside, hence we mostly use the term determinate.

Nondeterminism describes a situation where we do not have enough information to predict a specific behaviour. If we say an entity E generates `0`s and `1`s nondeterministically, we admit that we cannot predict what the next generated value will be. We cannot exclude any possibility, not even corner cases such as that no `1` is ever produced. If we require that E is *fair*, we constrain its behaviour and request that if E produces an infinite sequence, it must produce both infinitely many `0`s and infinitely many `1`s, thus it cannot block the generation of `1`s forever. To require that the generated sequence has a specific stochastic distribution, is a stronger constraint. It means not only that the number of `0`s and `1`s must be infinite in an infinite sequence, we also define the relative number of `0`s and `1`s. For instance if we request a uniform distribution we require that the ratio $\frac{\text{count}(0)}{\text{count}(1)}$ becomes `1` when the sequence length becomes infinite.

Nondeterminism allows to express uncertainties and different possible behaviours. In contrast a determinate model can only express one definite behaviour explicitly. Thus, nondeterminism increases the expressiveness of a modelling language. However, nondeterminism makes the task of analysis, formal verification, validation by simulation, and synthesis much more complex.

We argue that it is beneficial to use a stochastic distribution instead of nondeterminism for three reasons. First, very often we know more about an entity than nondeterminism would suggest. E.g. we know that its behaviour follows a specific probabilistic distribution. The additional information can be useful for simulation, synthesis and validation. Second, stochastic behaviour can be modeled by means of a pseudo random generator, which essentially is determinate. In this way we can avoid some of the difficulties that come with nondeterminism, but increase the expressiveness as compared to determinate models. Third, nondeterminate behaviour can be approximated by stochastic behaviour. In particular fair nondeterminism can be approximated rather well by a stochastic process. We acknowledge that nondeterminism cannot be fully simulated by a stochastic process but we have not yet encountered a concrete example where it is an advantage to use a nondeterminate model rather than a stochastic one.

In the next section we trace the history of determinate and nondeterminate models which forms the context for our proposal to use a stochastic process to get *almost* the expressiveness of nondeterminate models and to keep *almost* all the advantages of full determinism. Then we introduce some basic concepts of the Formal System Design ($\mathcal{F}or\mathcal{S}y\mathcal{D}e$) methodology which sets the context for the discussion thereafter. In section 4 we introduce the stochastic σ process. In section 5 we discuss transformations which refine stochastic processes into non-stochastic, determinate processes. Finally we conclude the paper in section 6 with a general discussion on the use of stochastic processes.

2. RELATED WORK

Submitted to CODES 2001

2.1 The Descriptive Purpose

In theoretical computer science nondeterminism has received continuous attention over decades due to the difficulties to deal with it in a satisfactory manner. On one hand nondeterminism has been considered mandatory as a modelling concept when writing distributed programs. When these programs are compiled and executed on a particular machine, the delays of computation and communication depend on the details of the target machine. If the different delays potentially lead to different behaviour, the abstract program is nondeterminate. Hence, nondeterminism is used to capture the timing behaviour of the target machine.

2.1.1 Determinate Models

However, the inclusion of nondeterminism severely complicates the attempt to define a precise semantics for a computer program. One track of research has therefore excluded nondeterminism by defining the semantics of a language in such a way, that its behaviour is independent of the execution delays of the target machine. In Kahn's language for parallel programs [7] both the individual processes as well as an composition of processes are determinate functions over infinite input streams. Kahn's semantic is very elegant and useful and had long lasting influence on various research directions and application fields. But the restriction that he imposed for the sake of determinate behaviour sometimes impede the programmer to formulate more efficient solutions to a problem. For instance one restriction in Kahn's language is that processes cannot test for the emptiness of an input channel, a feature known as "blocking read". Often it is obvious to a programmer that resources are better utilized if a process may check if input data is available and do something else if it is not.

While Kahn process networks and its descendants, e.g. data flow networks, took the approach to define a behaviour which is independent of timing properties, the perfectly synchronous languages [1] impose on any implementation the constraint that it has to be "fast enough". For programs and their implementations which fulfill this assumption, the behaviour is determinate, again by separating timing properties from the behaviour.

Clocked synchronous models have been used in hardware design to achieve the same. A circuit behaviour can be described determinately independent of the detailed timing of gates, by separating combinatorial blocks from each other with clocked registers. An implementation will have the same behaviour as the abstract circuit description under the assumption that all combinatorial blocks are "fast enough". This assumption has been successfully used for design, synthesis and formal verification of circuits.

In summary, determinate models achieve determinate behaviour by separating timing properties from behavioural properties. Variants of Kahn process networks define the semantics such, that any behaviourally correct implementation is acceptable independent of its timing. On the other hand perfectly synchronous and clocked synchronous models divide possible implementations into two classes, those which are "fast enough" are acceptable and those which are "too slow" are not acceptable.

2.1.2 Nondeterminate Models

Nondeterminism has been studied in data flow networks with asynchronous, infinitely buffered communication and in process algebras with synchronous, unbuffered communication.

One approach to generalize Kahn's theory for nondeterminate process networks is to use history relations rather than history functions. A history relation maps an input stream onto a set of possible output streams instead of a single determinately defined output stream. However, history relations are not sufficient to model nondeterminism. Brock and Ackerman [2] showed with examples that two components with identical history relations, if placed in the same context of a bigger system, may cause the system to behave in a different way, i.e. the system has different history relations. This means, that history relations are not sufficient to capture all relevant information about a component. In particular causality information between events must be included. Brock [2, 3] gave a formal semantics based on history relations and scenarios, which represent causality. Kosinski [8] described a semantics for nondeterminate data flow programs based on the idea to annotate each event with the sequence of nondeterminate choices that leads to that event. Park's formal semantics of data flow [10] models nondeterminism with oracles. Each nondeterminate merge operator is provided with an extra argument called the oracle. It is a random sequence and controls from which input stream the next token for the output stream is selected. We follow this idea to some extent but replace the nondeterminate oracle with a stochastic oracle.

Hoare's CSP (Communicating Sequential Processes) [5] and Milner's CCS (Calculus of Communicating Systems) [9] have been developed in response to two difficulties with data flow models. First, it appeared difficult to find elegant solutions for the formal semantics of nondeterminate data flow languages. Second, data flow models require unbounded buffers for communication which lead to difficulties in implementation. In almost the entire work on process algebras nondeterminism is assumed and significant effort has been spent to establish properties, equalities, and methods to guarantee a specific behaviour of the system in the presence of nondeterminism. For CCS, Milner defines the concept of *weak determinacy* [9], which is based on observational equivalence. A system in a given state with given inputs can enter a set of different successor states nondeterministically. If the system in all successor states behaves identically, as far as it can be observed from the outside, the system is weakly determinate. A somewhat broader concept is *weak confluence*. A system is confluent if for every two possible actions, the occurrence of one can never preclude the other. Thus, even though one of the two is selected nondeterministically the other will eventually also occur. Milner then gives a set of construction rules which preserve confluence [9].

In summary, determinate models restrict the model to guarantee a well defined behaviour in the presence of nondeterministic mechanisms. To the same end the construction process is constrained for nondeterminate models. Nondeterministic mechanisms are part of the implementation realm for the determinate models and part of the models themselves for the nondeterminate models.

2.2 The Constraining Purpose

For the purpose of describing requirements on a system various techniques related to nondeterminism have been used. Relations divide the possible responses of a system to a given

input into two parts: those acceptable and those not acceptable. `execution_time(Program)<5ms` and `size(Chip)<1cm`^{2} are two relations constraining the nonfunctional properties of a system. The relation defining a sorted integer array has been used numerous times as an example of a functional requirement. Dennis and Gao [4] describe the example of a transaction server, which accepts requests at several inputs and processes them (figure 1). The merge process, which

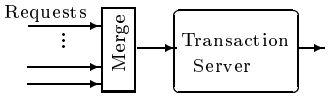

Figure 1: A transaction server with request merger.

decides the order in which requests are served, is subject to several functional and nonfunctional constraints. Apparently each request should eventually be served. Perhaps we require that the average response time is similar for requests on all input lines. And most likely we would like to have a high performance of the merge operator itself while minimizing its implementation cost. Clearly, the functional specification should not define a determinate merge mechanism to allow the allow the design process to find the best solution. The specification should rather be content with defining all allowed behaviours.

While relations constrain the functionality, perfectly synchronous and clocked synchronous models constrain the timing behaviour. With respect to functionality these models are fully determinate, hence we have discussed them in section *2.1.1*. However, with respect to timing they constrain the implementation to be "fast enough".

3. *ForSyDe* METHODOLOGY

In order to make the features of stochastic σ processes (section 4) understandable we need to introduce a few key concepts of the Formal System Design (*ForSyDe*) Methodology for which we have developed the σ process. *ForSyDe* [12, 13] is a fully determinate system specification and modelling technique. It adopts the perfectly synchronous assumption that neither communication nor computation takes time. It employs skeletons, which give the system description a structure, separate functionality from timing, and have explicit interpretations for hardware and software implementations. Skeletons are templates for processes which are connected by streams. In the following section we introduce the σ processes and the *ForSyDe* skeletons which encapsulate them.

4. THE σ PROCESS

A σ process is a pseudo random generator with a defined statistical distribution. We use it in two ways.

First, we use it to constrain the implementation of the system with respect to behaviour. Depending on the precise kind of the σ process, the implementation may or may not be required to respect the statistical properties of the specifying process. Note, that we do not use the σ process to constrain the timing behaviour. For the timing behaviour we strictly follow the approach of the synchronous languages by requiring that the implementation is "fast enough" [13].

Second, environment elements can be modeled with σ processes when we cannot or do not want to represent their exact behaviour or timing. We think a statistical distribution is more appropriate than nondeterminism. Consider an ATM switch which receives ATM cells from the environment. The type of ATM cells, user cells, alarm cells, maintenance cells, erroneous cells, etc., follow a statistical distribution. To generate ATM cells according to given probabilities is both more accurate and more useful for the design and validation of the ATM switch.

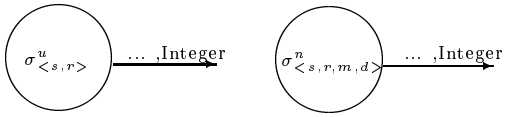

Figure 2: The σ processes for uniform and normal distributions. s denotes the seed value, r the range, m the mean value, and d the standard deviation.

A σ process is instantiated with a few parameters which define the statistical properties of the generated infinite stream of integers (figure 2). The σ^u process generates a uniform distribution within a given range r and the σ^n process generates a normal distribution defined by the mean value m and the standard deviation d. σ processes for other distributions can be defined as needed. The σ processes are true functions because they use a pseudo random generator which is initialized with a specific seed value s. For the sake of simplicity we use only σ^u processes in the rest of the paper.

In the *ForSyDe* methodology we assign different interpretations to the σ processes, depending on the design activity. The description that we have given above is the interpretation for simulation. For synthesis and verification we adopt a different interpretation. We distinguish two variants:

A synthesized *sigma bar process* $\bar{\sigma}$ can generate any of the possible outputs of a $\bar{\sigma}$ process without restriction. The output of the synthesized process may or may not have the statistical properties of the specification process. For instance a synthesized $\bar{\sigma}^u_{<s,[0,1]>}$ could be implemented in such a way, that it generates always a 0 and never a 1. On the other hand *sigma tilde processes* $\tilde{\sigma}$ have to be implemented such that the statistical properties are preserved.

The merge of the transaction server in figure 1 definitely requires a $\tilde{\sigma}$ process. However, consider the `bar-merge` operation illustrated in figure 3. In each processing step the

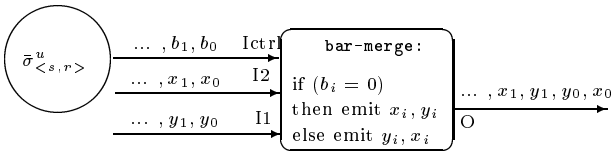

Figure 3: A merge which uses a $\bar{\sigma}$ process.

bar merge receives one token from each of its two inputs I1 and I2 and emits the two tokens in arbitrary order. Since we do not want to specify the order of the two output tokens deterministically, we use a $\bar{\sigma}$ process to drive the third input of the `bar-merge`. If that token is a 0, first an x and then a y is emitted; if it is a 1, the tokens are emitted in reversed order. For the implementation we are free to select any order; we may select a hardwired solution to always emit x before y.

Formal verification and analysis is only allowed to use the statistical properties of the sequences generated by σ, which are guaranteed by any implementation. Hence, formal analysis follows the interpretation of synthesis.

If desirable, more process types with different interpretations can be defined. For instance it should be investigated if a σ process type with a fair synthesis and analysis interpretation is useful.

4.1 *ForSyDe* Skeletons

Strictly speaking the *ForSyDe* models are still fully determinate because σ processes are based on pseudo random generators to produce sequences with specific statistical properties. Then we request synthesis and verification tools to respect only these statistical properties and to ignore the specific values of those sequences.

The σ processes are not directly visible to the user. *ForSyDe* is based on skeletons which provide the system structure and the system timing. Furthermore, skeletons have specific interpretations in the context of design and implementation which allow for an efficient, template driven synthesis [12]. Consequently, we use skeletons to encapsulate the σ processes.

In the following we discuss one simple *ForSyDe* skeleton, namely `mapS`. `mapS` repeatedly applies a combinatorial function on individual values of the input stream and thus produces the consecutive values of the output stream. Now we develop a variant which also contains a σ process. We can do this in two different ways. First, we can use a `select` operator which applies one out of two functions on an input value depending on the result of a σ process. Second, we can internalize the choice into the combinatorial functions. In this paper we only present the first option.

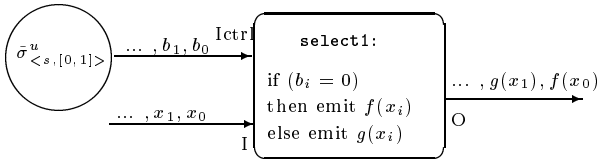

Figure 4: The `selMapS` skeleton.

The skeleton `selMapS` uses the `select` operator and a σ process `sigma` to apply one out of two functions repeatedly on the values of the input stream (figure 4). `select`(x, b, f, g) evaluates to $f(x)$ if $b = 0$, and to $g(x)$ if $b = 1$. `selMapS` takes four instantiation parameters. f and g are the two functions which are possibly applied to the values of the input stream. s is the seed for `sigma`, and `sigmatype` defines if `sigma` is a bar or a tilde process.

Stochastic variants can be developed for many other *ForSyDe* skeletons in a similar way.

5. TRANSFORMATIONS

ForSyDe is a transformation based methodology [14], which distinguishes between semantic preserving and decision making transformations. When a σ process is refined into an implementation, typically a design decision is made. In this section we give examples of transformations that transform a σ process into a non-stochastic implementation. As we will see, each transformation embodies a particular design decision.

For the sake of conciseness we do not use the *ForSyDe* language but an abbreviated notation. As example we use a

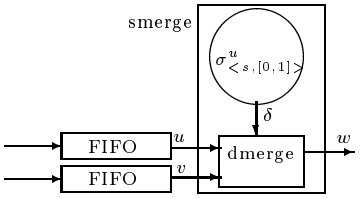

Figure 5: A merge process with a stochastic oracle.

simple merge process, which is borrowed from D. Park [10] and adopted to our needs. Both inputs are buffered with infinite FIFOs as illustrated in figure 5. The buffers prevent loss of data if there is temporarily more data on the two inputs than can be emitted to the output. We assume the buffers are infinite to concentrate on the merge operation itself. We use the following conventions: f and g are determinate functions implementing the merge operation. u, v and w are infinite sequences at the inputs and outputs. a and b are individual data values in these streams. We use the `dot` notation to concatenate values and streams, e.g. $a.u$ is a value a followed by the stream u. $\perp$ is a token in the stream indicating the absence of a value. Remember that we use a perfectly synchronous model, which means that we can detect when no data is available at a particular time instance. The FIFOs do not store $\perp$ tokens but an empty FIFO emits a $\perp$ at each time instance. δ is the infinite streams of `0`s and `1`s coming from the σ process. With this we define the `dmerge` of figure 5 as follows.

$$\texttt{dmerge} \equiv f(a.u, v, 0.\delta) = f(u, a.v, 1.\delta) = a.f(u, v, \delta)$$

`dmerge` takes the first token from one of the input streams, depending on the current value from the σ process, and emits it to the output. The stochastic merge `smerge` contains `dmerge` and the σ process.

Strict round robin: One possible implementation of `smerge` is strict round robin as defined by `strictRR`:

$$\begin{aligned} \texttt{strictRR} \equiv \quad & f(a.u, v) = a.g(u.v) \\ & g(u, b.v) = b.f(u, v) \end{aligned}$$

`strictRR` transmits even $\perp$ tokens. This may be no problem if the load on both input streams is on average less than or equal to half the bandwidth on the output stream. If this is not the case our merge process cannot handle the inputs quickly enough, even if the combined load on both input streams is less then the bandwidth of the output stream. The reason is that we dedicate a full time slot alternately to each input stream independent if there is a value or not.

The *ForSyDe* transformation rule for transforming a `smerge` into a `strictRR` merge is

$$\texttt{smerge\_strictRR} = [\,] \models \texttt{smerge} \Leftrightarrow \texttt{strictRR}$$

`smerge_strictRR` is the name of the rule. The expression in square brackets is the premise of the rule, which in this case is empty because `strictRR` is equivalent to `smerge`. Let w_n be the n^{th} element in stream w, and let $P_v(w_n)$ be the probability that $w_n \in v$. Then it is easy to see that $P_v(w_n) = P_u(w_n) = 0.5$ for both `smerge` and `strictRR`.

$\perp$-sensitive round robin: If above solution is not acceptable we can adopt a round robin procedure which is

sensitive to the absence of values.

$$\texttt{sensitiveRR} \equiv \begin{array}{ll} f(\perp .u, b.v) = b.g(u,v) & \\ f(a.u, v) = a.g(u,v) & a \neq \perp \\ g(a.u, \perp .v) = a.f(u,v) & \\ g(u, b.v) = b.f(u,v) & b \neq \perp \end{array}$$

The problem here is that we deviate from the behaviour of the `smerge` even with respect to the stochastic properties. We can still formulate a transformation from `smerge` to `sensitiveRR` but only under certain conditions. Let $P_0(\delta_n)$ be the probability that the n^{th} element of the δ-stream is `0`, and let $P_p(v_n)$ be the probability that the n^{th} element of v is a valid data, i.e. it is not $\perp$. Then we can formulate the transformation rule as follows.

$$\begin{array}{l} \texttt{smerge\_sensitiveRR} = \\ \qquad [\forall n : P_0(\delta_n) \geq P_p(v_n) \wedge P_1(\delta_n) \geq P_p(u_n)] \\ \qquad \models \texttt{smerge} \Leftrightarrow \texttt{sensitiveRR} \end{array}$$

The premise here relates the stochastic properties of the σ process to the stochastic properties of the input streams u and v. For instance, if the σ process has a uniform distribution as in figure 5, the load on both inputs must be less than or equal to 50%, which is equivalent to the requirement that $\forall n : P_p(v_n) \leq 0.5 \wedge P_p(u_n) \leq 0.5$. Consequently, if we expect different loads on the input streams, we have to adapt the σ process accordingly in order to maintain the same stochastic properties for specification and implementation. This will work as long as the combined load of the input streams is less or equal to the maximum load on the output stream.

Unfair arbiter: The `sensitiveRR` may be a more efficient solution but not the best one for all situations. Assume we have to maximize throughput and we have to pay a high delay penalty when we switch from one input stream to the other. On the other hand, our FIFOs are sufficiently long to buffer even long bursts. For this the unfair arbiter `unfairA` might be a preferable solution.

$$\texttt{unfairA} \equiv \begin{array}{ll} f(\perp .u, b.v) = b.g(u,v) & \\ f(a.u, v) = a.f(u,v) & a \neq \perp \\ g(a.u, \perp .v) = a.f(u,v) & \\ g(u, b.v) = b.g(u,v) & b \neq \perp \end{array}$$

`unfairA` transmits data from one input as long as there is input. It only switches to the other input when a $\perp$ is encountered.

The corresponding transformation rule is

$$\begin{array}{l} \texttt{smerge\_unfairA} = \\ \qquad [\forall n : P_0(\delta_n) \geq P_p(v_n) \wedge P_1(\delta_n) \geq P_p(u_n)] \\ \qquad \models \texttt{dmerge} \Leftrightarrow \texttt{sensitiveRR} \end{array}$$

Note, that the premise here is the same as for `sensitiveRR`, because we assume infinite buffers at the inputs of the merge. However, the choice of the merge implementation has an impact on the required buffer size. For the sake of brevity and comprehensiveness of this discussion we have assumed infinite buffers, which allows us to operate with stochastic properties on infinite streams. To select a finite buffer size for the FIFOs we must consider stochastic properties of finite subsets of the infinite streams. This is beyond the scope of this paper but we would like to emphasize, that the basic principle is the same, which is to relate a design decision to stochastic properties of the environment. These relations are expressed in the premises of transformation rules as exemplified above. In our methodology we use the premises as documentation and, partially, as proof obligations. They document the design decisions taken, under which conditions they are allowed, and how these decisions constrain other parts of our system or the environment. If the premises constrain other parts of our system, e.g. if the input streams to the merge originate within our system, we have to prove that the affected parts comply with these constraints. In this way stochastic properties are propagated through the system until they can be firmly based on properties of the environment.

6. USAGE OF STOCHASTIC SKELETONS

In the introduction we have stated that we use the stochastic skeletons with similar objectives as nondeterminism. We have also discussed briefly how different design activities should interpret these skeletons. Now we summarize our objectives and delineate them from issues that we do not intend to address.

6.1 The Descriptive Purpose

One important application of stochastic skeletons is to model environment components. Very often we do not know the exact behaviour of the environment, or that we do not care about all details. The uncertainty about the environment concerns both the functional as well as the timing behaviour. With nondeterminism we allow different environment behaviours and we give no further information about how likely the different cases are. If we add a notion of fairness we exclude a few specifically undesirable possibilities. However, very often we know a bit more and we would like to assign probabilities to the different possible behaviours. The simulations that we can perform based on probabilistic environment behaviour will be often more realistic.

We have to acknowledge though, that the notion of nondeterminism is broader than any specific probability distribution, because it encompasses all possible probabilistic distributions. A nondeterminate process may generate sequences of numbers which exhibit any possible probability distribution, while a stochastic process can only generate sequences with a given probability distribution. However, for two reasons we doubt that this difference is of practical importance. First, any sequence that a nondeterminate process can generate, can also be produced by a stochastic process. Second, all implementations of simulators which allow to simulate nondeterminate behaviour, use in fact some stochastic process for this. Our computer technology does not allow us to simulate nondeterminate behaviour which does not follow a specific probability distribution. These arguments justify the substitution of a nondeterminate process by a stochastic process. However, we go even further and claim that it is an advantage to do this. If a user includes a nondeterminate process in a model, she has no control over what the implementation of the simulator is in fact doing. Since a nondeterminate process may generate any sequence of numbers, the implementor of the simulator typically selects one of the possibilities which is most convenient for her. But this decision is often unknown to the end user who uses the simulator. On the other hand the simulator must respect defined statistical properties. Thus, the end user has a better control over the behaviour of the model. Furthermore, by using pseudo random generators a particular simulation run is repeatable which greatly helps debugging.

While we advocate strongly the usage of stochastic pro-

cesses for the description of the environment, we do not propose them for describing uncertainties of the system under design. In the context of specification we find the notion, that we do not know the exact behaviour, with respect to timing or function, not satisfactory. We design, implement and manufacture the system. So in principle we have full control and knowledge about it if we decide to dedicate the necessary effort. Sometimes we may not want to spend the effort because we do not care as long as the behaviour falls into a given class or range of acceptable behaviours. Consequently, we prefer the notion that we *constrain the system*. We do this although we acknowledge that in software development the uncertainty about timing properties of the underlying hardware machine was the foremost motivation for nondeterminism. However, we do not address problems of general software development but we discuss the specification and modelling of embedded systems, hardware and embedded software, where we always are concerned with performance issues and we often face hard timing constraints. Therefore we find the concept of constraining the timing behaviour of the implementation more appealing than the assumption, that the timing behaviour could be arbitrary.

6.2 The Constraining Purpose

The functional and timing behaviour of a system implementation can be constrained in a variety of ways and stochastic processes should only be used in some specific cases. The *ForSyDe* methodology uses a perfectly synchronous timing model, which is very well suited to constrain the timing behaviour. Both in hardware and in embedded software design, a synchronous design style has been used with great success. It effectively separates functionality from timing issues. Static timing analysis can be done independently from functional validation and verification. A rich set of pipelining and retiming techniques have been developed to tune the timing behaviour while keeping the functionality.

The general method to express constraints on the functionality is by means of relations. In the early phase of system development, the requirements analysis phase, general requirements and constraints are formulated in terms of relations. However, because relations allow a huge design space, efficient synthesis techniques. A system specification model, which captures most of the high level design decisions, is therefor a necessity [6]. This model should be determinate because nondeterminism greatly complicates synthesis and validation.

However, as we tried to illustrate in several examples in this paper, there are occasions when we would prefer to leave several options open in order to give the later design phases more opportunities to find optimal implementations. Stochastic processes are a good way to address this issue. For simulation they allow to exercise all possibilities which might occur in a concrete implementation. Synthesis can exploit the possibilities that a stochastic process exposes. Validation can use their statistical properties to verify system properties.

7. CONCLUSION

We have thoroughly reviewed the history of determinate and nondeterminate models to reveal the inherent trade-off involved. Determinate models are significantly easier to analyze, verify and synthesize. On the other hand nondeterminate models are more expressive. For the specific purpose of specification of embedded mixed hardware/software systems we propose to use stochastic processes to approximate the expressiveness of nondeterminate models while preserving much of the analytic capabilities for determinate models.

8. REFERENCES

[1] Albert Benveniste and Gérard Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.

[2] J. Dean Brock and William B. Ackerman. Scenarios: A model of non-determinate computation. In J. Diaz and I. Ramos, editors, *Formalism of Programming Concepts*, volume 107 of *Lecture Notes in Computer Science*, pages 252–259. Springer Verlag, 1981.

[3] Jarvis Dean Brock. *A Formal Model for Non-deterministic Dataflow Computation*. PhD thesis, Massachusets Institute of Technology, 1983.

[4] Jack B. Dennis and Guang R. Gao. Multiprocessor implementation of nondeterminate computation in a functional programming framework. Technical Report Computation Structures Group Memo 375, Laboratory for Computer Science, Massachusetts Institute of Technology, January 1995.

[5] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–676, August 1978.

[6] Axel Jantsch and Ingo Sander. On the roles of functions and objects in system specification. In *Proceedings of the International Workshop on Hardware/Software Codesign*, 2000.

[7] Gilles Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress 74*. North-Holland, 1974.

[8] Paul R. Kosinski. A straight forward denotational semantics for nondeterminate data flow programs. In *Proceedings of the* 5^{th} *ACM Symposium on Pronciples of Programming Languages*, pages 214–219, 1978.

[9] Robin Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[10] David Park. The 'fairness' problem and nondeterministic computing networks. In De Baker and van Leeuwen, editors, *Foundations of Computer Science IV, Part 2: Semantics and Logic*, volume 159, pages 133–161. Mathematical Centre Tracts, 1983.

[11] Ingo Sander and Axel Jantsch. Formal design based on the synchronous approach, functional models and skeletons. In *Proceedings of the Twelfth International Conference on VLSI Design*, 1999.

[12] Ingo Sander and Axel Jantsch. System synthesis based on a formal computational model and skeletons. In *Proceedings of the IEEE Computer Society Annual Workshop on VLSI*, 1999.

[13] Ingo Sander and Axel Jantsch. System synthesis utilizing a layered functional model. In *Proceedings of the 7th International Workshop on Hardware/Software Codesign*, pages 136–141, May 1999.

[14] Wenbiao Wu, Ingo Sander, and Axel Jantsch. Transformational system design based on a formal computational model and skeletons. In *Proceedings of the Forum on Design Languages*, September 2000.