

# Representation of Function Variants for Embedded System Optimization and Synthesis

K. Richter, D. Ziegenbein, R. Ernst\*  
IDA / TU Braunschweig  
D-38106 Braunschweig / Germany  
{richter|ziegenbein|ernst}@ida.ing.tu-bs.de

L. Thiele  
TIK / ETH Zürich  
CH-8092 Zürich / Switzerland  
thiele@tik.ee.ethz.ch

J. Teich  
DATE / UNI Paderborn  
D-33098 Paderborn / Germany  
teich@date.uni-paderborn.de

## Abstract

*Many embedded systems are implemented with a set of alternative function variants to adapt the system to different applications or environments. This paper proposes a novel approach for the coherent representation and selection of function variants in the different phases of the design process. In this context, the modeling of re-configuration of system parts is supported in a natural way. Using a real example from the video processing domain, the approach is explained and validated.*

## 1 Introduction

Many embedded systems are implemented with a fixed core function and a set of alternative *function variants* to adapt the system to different applications or environments. Examples are TV sets which can be adapted to different standards or automotive control systems to be used in countries with different emission laws. Function variants are mutually exclusive, i. e. only one variant of a set of alternative functions is selected a time. There may be several of those variant sets in one embedded system, e. g. for different input and output standards of a multi-media device. The variant selection for these sets may be related or independent.

Once a function variant is selected, it remains fixed throughout the operation of that system. In contrast, a system may have different *modes of operation* [1, 9] which are mutually exclusive, as well, but can be changed dynamically during system operation with a transition from one mode to another. As an example, call set-up, connected, disconnect, and stand-by could be modes of a mobile communication device. The distinctive feature of modes and variants is that modes and mode transitions are part of a single system function while function variants define alternative system functions without transitions between them.

With increasing embedded system complexity, system design will have to resort to configurable systems in order to reach the required design productivity and a sufficient market share to make large systems-on-a-chip profitable. The use of function variants will

be a key technique in this context. As a popular example, the Philips TriMedia processor already contains several coprocessors with pre-defined function variants for different standards and applications [7].

*Function variant selection* can occur at different stages of a product life time corresponding to different variant types. *Production variants* are selected at production time, e. g. by downloading a certain software variant into an EPROM. *Run-time variants* are selected at system start-up time, e. g. as part of a boot sequence which reads switches or flash memory stored parameters. In both cases, system optimization can assume that the system's variants can not be changed during system operation.

A more complicated selection process is found in dynamically *reconfigurable architectures*. Here, a subsystem is typically configured by a higher level controller to execute a function which the subsystem itself cannot change. In other words, what appears as a variant at the subsystem level becomes a system mode at the controller level. The same dependency can be found in programmable coprocessors, such as a filter processor (subsystem) which cannot change its own function but is controlled by a core processor [7]. A more elaborate discussion of system design with function variants and flexibility requirements can be found in [2].

Since function variants and, consequently, variant selection are not part of the (sub)system function, their representation should be treated independently. We will first focus on *function variant representation*. Function variants define a set of system behavior alternatives. On the other hand, system specification languages and high-level system representations typically describe a single system behavior, only, (e. g. [4]). Such design representations are appropriate to implement a single variant, but they are not able to capture the complete design with all variants, and therefore, are not sufficient for our purpose of system optimization.

So far, there are only few proposals in literature to include variants in system synthesis. In [6], all variants (in this case different applications) including timing constraints are enumerated and serialized into a single large task which is synthesized to a hardware, such that all timing constraints of all variants are met. This approach is based on a unified representation of all variants, but works for single process systems with single time constraints, only. In contrast, the authors in [5] use separate representations but serialize the design process by incrementally synthesizing the hardware architecture for one variant (application) at a time. Both groups report a dominant influence of the serialization order on result quality.

These serialization approaches assume a certain synthesis technique and are not easily applicable to more complex systems with multiple concurrent processes and multiple time constraints. Also, enumeration of system variants becomes infeasible with larger systems. Nevertheless, this research clearly demonstrated the feasibility and the cost benefits of synthesis regarding function variants.

\*This part of the work was supported by the German DFG.

The function variant representation should consider the variant selection mechanism to allow an implementation as run-time variants or with a reconfigurable architecture. The representation of the mechanism should support the reuse of a system part, possibly with a different function variant type. To give an example, a network protocol that has been implemented as a production variant in hardware might end up as a software-implemented run-time variant in the next product generation. Selection mechanisms have not been considered in previous work on system synthesis with function variants.

In summary, function variant representation is an unsolved problem. The few previous papers [6, 5] on this topic focused on synthesis with variants thereby tailoring the design representation to the specific synthesis algorithm and limited system model. Instead, the work presented here seeks a more general approach to capture function variants and variant selection in the design representation. Generality shall not come at the cost of additional design constraints such that model limitations resulting from the introduction of variants shall be minimized. This is a challenging task, and this paper will concentrate on the proposed solution while application to synthesis is deferred to later work.

As a basis, we use the SPI communicating process model [8, 9] which is specifically targeted to system synthesis. It is very suitable as a starting point offering a homogeneous data and control flow with local process modes which simplifies the representation of variant selection mechanisms. Moreover, SPI processes are represented by behavior intervals (upper and lower bounds on communicated data, timing, ...) which allows to combine many variants in a single abstract process in order to simplify the design process and support reuse. Most importantly, we could show previously [8, 9] that SPI can be used as a common representation for very different models of computation. This is a prerequisite for the goal of a more general approach.

After a short introduction of the SPI model, the paper will derive the model extensions to represent function variants and variant selection. The representation is, then, illustrated with some simplified examples. Finally, we will draw some conclusions.

## 2 The SPI Model

In this section, the concepts of the SPI (System Property Intervals) model [8, 9] necessary for the understanding of this paper are introduced. It has to be stressed that the main goal of this paper is not to specifically extend the SPI model but to introduce the concepts of function variant representation and selection. SPI has been chosen only to demonstrate the approach.

In the SPI model, the system is represented as a set of concurrent processes which communicate via unidirectional channels that are either FIFO-ordered queues (destructive read) or registers (destructive write). Such models are usually represented as directed, bipartite graphs. A *model graph* in SPI consists of process nodes ( $P$ ), channel nodes ( $C$ ) and directed communication edges ( $E$ ).

While each channel node simply transfers data from the sender to the receiver without any transformation, the functionality of process nodes can be of arbitrary complexity. But the exact internal behavior of the processes and functionality performed by them does not have to be known for the purpose of optimization at the process level. Thus, processes and channels are modeled only by their abstract external behavior. This behavior is captured by a small set of parameters which are extracted from the original specification and annotated to the graph nodes. The next paragraphs briefly review the parameters that are important in the context of this paper.

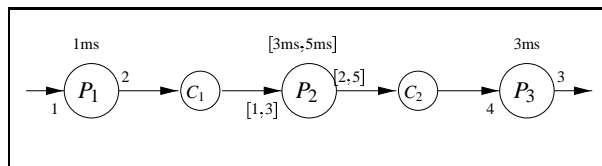


Figure 1: SPI Example

At each process execution, a process maps input data to output data. However, since we are not interested in the function performed by a process, the communicated data is only represented by the amount of data. For example, process  $p_1$  in Figure 1 consumes 1 data token and produces 2 data tokens at each execution. The latency of  $p_1$  (i. e. the difference between starting and completion time of  $p_1$ ) is 1ms. This process is completely determinate and all parameters are fixed in value. This is not necessarily the case for all processes, since the information about the behavior may be incomplete or uncertain due to a possibly data-dependent functionality or internal state. Process  $p_2$  for example is specified with intervals for the modeling parameters. Such intervals represent lower and upper bounds for the actual value of the parameter. Thus,  $p_2$  consumes at least 1 and at most 3 tokens from channel  $c_1$  and produces at least 2 and at most 5 tokens on channel  $c_2$ , respectively. The execution latency is between 3ms and 5ms.

Mostly, the parameters of a process are not independent from each other but strongly correlated. For a more accurate modeling, such correlation information can be specified by means of sets of process modes. Each mode thereby represents a subset of all possible process behaviors. For example, process  $p_2$  can be represented as having two alternative modes:

$$\begin{aligned} m_1 &= (3ms, 1, 2) \\ m_2 &= (5ms, 3, 5) \end{aligned}$$

Then e. g. in mode  $m_1$  process  $p_2$ 's latency is 3ms, it consumes 1 token and produces 2 tokens, etc. Nevertheless, without specifying rules for the selection of a mode, the behavior of process  $p_2$  is still uncertain since  $p_2$  may execute in  $m_1$  or in  $m_2$ .

Usually, processes adapt their behavior based on the contents of the consumed data. Since data is previously represented by its amount only, processes may add *virtual mode tags* to produced data tokens to visualize certain content information. Then, the receiving process can select a particular mode depending on the availability of certain mode tags. Therefore, an *activation function* is associated with each process that may be formulated as a set of rules. These rules map input token predicates to modes. A predicate in this case is either 'true' or 'false' depending on the number of tokens and the tag set of the *first visible* token on the input channels of the process. For process  $p_2$  from the above example, these rules could be:

$$\begin{aligned} a_1 &: (c_1.num \geq 1) \wedge ('a' \in c_1.tag) \mapsto m_1 \\ a_2 &: (c_1.num \geq 3) \wedge ('b' \in c_1.tag) \mapsto m_2 \end{aligned}$$

Assuming that process  $p_1$  adds one of the tags 'a' or 'b' to the tag set of all produced tokens, the behavior of  $p_2$  is completely determinate. If there is at least 1 available token on channel  $c_1$  and if the tag 'a' is included in the tag set of this token, process  $p_2$  is activated in mode  $m_1$ . Analogously, if there are at least 3 tokens available on  $c_1$  and the first one has 'b' in its tag set,  $p_2$  is activated in mode  $m_2$ . Obviously, if there is not tag on the first visible token on channel  $c_1$ , no activation rule is enabled and the process is not activated. Such situations can be ignored due to the assumption of correct models.

Above, the SPI model has been reviewed as precise as necessary for the understanding of this paper. However, the SPI model has been formally defined in [8, 9]. Update rules have been introduced to formally show how modelings evolve over time. Furthermore, the concept of virtuality for processes and channels enables to model the system and the environment coherently. Timing constraints have been defined as well as a constructive method to check their compliance. It has been shown that the SPI model is capable to capture the needed information for scheduling and allocation from static and dynamic data flow models, hardware description languages, real time operating system's process models and state-based models.

### 3 Function Variants

After reviewing the basic ideas of SPI, the following sections contain the novel aspects of this paper, that is the modeling of *function variants* and *dynamic variant selection*. As already mentioned in the introduction, systems may differ in parts of the functionality while having other parts in common, e. g. multi-standard TV sets. Instead of one single modeling for each of the system's variants, all variants need to be represented in a complete modeling to support overall system design optimization. Therefore, the SPI model is extended by a few simple but powerful constructs.

As mentioned in the previous section, all functionality resides inside the nodes while edges define the interaction structure. Usually, it can be assumed that cohering functional parts of a system description are mapped to connected subgraphs in SPI. Thus, changing a system's variant in the functional description corresponds to exchanging subgraphs in SPI. Such subgraphs are usually represented as *clusters*. A cluster contains a set of graph elements which communicate through the cluster border via input and output ports.

#### Definition 1 (Cluster)

A cluster is a tuple  $\gamma = (I, O, P, C, E, \Psi)$  where  $I$  denotes the set of input ports and  $O$  the set of output ports, respectively.  $P$  denotes the set of embedded processes,  $C$  the embedded channels, and  $E \subseteq ((P \cup \Psi) \times (C \cup O)) \cup ((C \cup I) \times (P \cup \Psi))$  the embedded edges.  $\Psi$  denotes the set of embedded interfaces to be defined later. The out-degree of input-ports and channels and the in-degree of output ports and channels is at most one.

Clustering does not add functionality to the model and is only a structuring concept. The only restriction in this context is that a cluster, like a process, can only be connected to channels.

Now, a system with two function variants can be represented consisting of three parts. The first common part contains all elements that are not variant-dependent, while the remaining parts are mutually exclusive clusters which represent the distinct function variants. Evidently, both clusters must have the same external connections in terms of input and output ports, since otherwise they could not be reasonably exchanged by each other. In other words, the three parts need a common *interface*.

#### Definition 2 (Interface)

An interface is a tuple  $\Psi = (I, O, \Gamma)$  where  $I$  denotes the set of input ports,  $O$  the set of output ports, and  $\Gamma$  the set of clusters associated with this interface. Each cluster matches the interface in terms of input and output ports.

Using these two constructs, a system part for which different function variants exist can be represented by an interface with a set of different clusters associated. Then, each function variant is

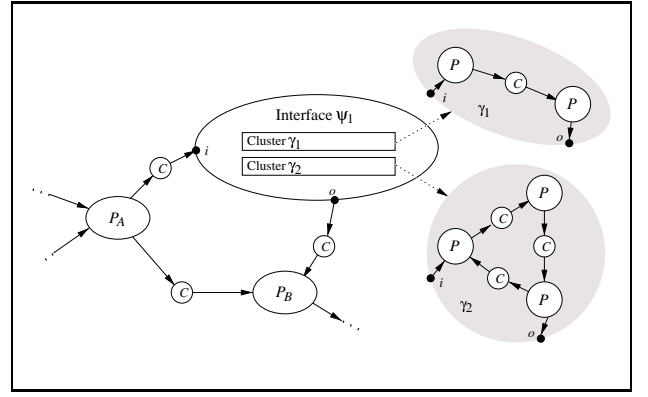


Figure 2: Example System with Two Function Variants

represented by exactly one of the clusters. An example for a system part with two function variants is the interface  $\Psi_1$  in Figure 2. The different variants are depicted as clusters  $\gamma_1$  and  $\gamma_2$ .

### 4 Function Variant Selection

So far, concepts have been introduced to represent mutually exclusive function variants in SPI. In this section, the selection mechanisms of the three different types of function variants are explained.

Production variants are selected by the designer before the actual run-time of the system, i. e. at production time. The final product implements one single function variant only without any selection capabilities. Clearly, this selection type is not part of the system's functionality and does not have to be modeled.

As already mentioned in the introduction, a more complicated selection process can be found in reconfigurable architectures. Here, a subsystem is typically configured by a higher level controller to execute a function which the subsystem itself cannot change. As previously mentioned, different function variants are represented by different clusters for one interface. Thus, the function variant selection corresponds to a cluster selection at the interface border, triggered by incoming data. Analogous to the mode selection mechanism for processes, we therefore associate a cluster selection function with interfaces. When a new configuration is selected, a configuration step must be inserted. This step consumes time, the configuration latency, that has to be considered. Additionally, we may want to access information about the currently selected cluster. Thus, this must be kept by means of an additional parameter.

#### Definition 3 (Cluster Selection)

Associated with each interface  $\Psi$ , there may be a Cluster Selection Function that may be formulated as a finite set of rules  $\Delta$  where each rule is a mapping of an input token predicate to one dedicated cluster. The predicate of a rule  $\delta \in \Delta$  is a function on the tag sets  $c.tag$  of the first available token on some input channels  $c \in Inputs(\Psi)$ . The value of the predicate is either 'true' or 'false'.

Associated with each interface  $\Psi$  and each cluster  $\gamma \in \Gamma_\Psi$  there is a configuration latency  $t_{conf}$  that denotes the time needed to configure interface  $\Psi$  with cluster  $\gamma$ .

Associated with each interface  $\Psi$  there may be a parameter  $\gamma_{cur} \in \Gamma_\Psi$  that denotes the currently selected cluster of interface  $\Psi$  at a given point in time.

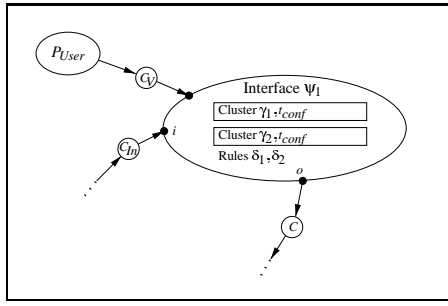


Figure 3: Selection of Run-Time Variants

Run-time variants are selected once at system start-up. One parameter, predefined or set by a user, is evaluated and one particular cluster is selected by the interface. An example for the selection of run-time variants is depicted in Figure 3. Process  $P_{User}$  models the user who selects the function variant. It writes a token on channel  $C_V$  that has an associated tag which is either 'V 1' or 'V 2' indicating the desired function variant. This tag is evaluated by the cluster selection rules of the interface and the interface is replaced by the corresponding cluster. The cluster selection rules are:

$$\begin{aligned} \delta_1 &: ('V 1' \in C_V.tag) \mapsto \gamma_1 \\ \delta_2 &: ('V 2' \in C_V.tag) \mapsto \gamma_2 \end{aligned}$$

To keep the examples understandable we omitted certain modeling elements needed to constrain the behavior of some system parts, in this case  $P_{User}$  to execute only once in the beginning.

In contrast to these run-time variants which are selected once and then remain fixed throughout the time of system operation, the dynamic selection of variants typically requires the replacement of clusters at run-time. This situation can be detected by comparing the newly selected cluster with the current one, stored in parameter  $\gamma_{cur}$ . Since parts of the cluster to be replaced may be in execution, this may include terminating the running cluster and then instantiating the new cluster. Evidently, the termination of a running cluster results in the loss of all data on the internal channels. Although this might be acceptable in certain situations, it may not be desired in others, e. g. when the selection of a new protocol for a communication device must be delayed until the preceding block has been correctly transmitted. Hence, clusters may sometimes require to complete part of their functionality before they may be terminated. In other words, the functionally coherent execution of a cluster as a whole may need to be guaranteed for a correct behavior of the entire system. The possible termination delay has to be accounted for in the corresponding configuration latency.

The approach we propose in this paper is to abstract clusters to processes and to use the concept of process modes to represent dynamic function variant selection. This way, we benefit from the similarities between the concept of interfaces with dynamically changing clusters and processes with mutually exclusive modes and dynamic mode selection depending on incoming data. Therefore, the set of clusters has to be mapped to a set of process modes. Obviously, the process parameters have to be extracted from the original specification of the clusters in consideration of the selection function of the interface. These parameters are latency time, production and consumption rates and the activation function. Depending on the contents of the clusters and the cluster selection function, this extraction process can be complex, it may even include the mapping of a single cluster to several modes. Thereby, additional designer knowledge allows abstraction at different levels of detail to particularly focus on the significant points, e. g. some situations of cluster

termination may be less critical than others. Then, the process mode determines the selected cluster and, thus, the selected function variant. Additionally, the parameter extraction should be carried out carefully, not to construct any pitfalls like dead locks, etc.

So far, we have not accounted for reconfiguration, i. e. the change from one function variant to another. This happens, when the modes of two consecutive process executions have been extracted from different clusters. In this case, a reconfiguration step must be inserted between these process executions. As previously mentioned, this step consumes time, the reconfiguration latency, that has to be considered. Because modes are not stored in a process, this situation can not be detected unless the current configuration (or cluster) is stored and annotated to the process. Therefore, the concept of configurations is defined.

#### Definition 4 (Configurations)

Associated with each process  $p \in P$ , there may be a set of configurations  $CONF = \{conf_1, \dots, conf_n\}$  where  $n$  is the number of function variants or clusters of process  $p$ ,  $conf = \{m_1, \dots, m_j\}$  is a set of process modes  $m$  and  $j$  the number of modes in one configuration  $conf$ . All modes within one configuration are extracted from the same function variant or cluster.

Associated with each process  $p \in P$  and each configuration  $conf \in CONF$ , there is a (re)configuration latency  $t_{conf}$  that denotes the time needed to configure process  $p$  with configuration  $conf$ .

Associated with each process  $p \in P$ , there may be a parameter  $conf_{cur} \in CONF$  that denotes the current configuration of process  $p$  at a given point in time.

At the subsystem level, it can be analyzed whether a newly activated mode belongs to the current configuration and, thus, to the current function variant, or not. If not, a new configuration is selected according to the configuration set of the process. The value of  $conf_{cur}$  is updated and the process is reconfigured. Thereby, the old configuration is destroyed including all internal buffers. After the reconfiguration latency, the process is executed in the newly activated mode. From the higher level point of view, the reconfiguration latency is simply added to the process execution latency for this execution.

As an example, we apply this procedure to our example system on Figure 3. Therefore, we replace the interface  $\Psi_1$  by a process  $P_{Var}$  with two associated configurations, namely  $conf_1$  and  $conf_2$ . The measured latencies are  $t_{conf;\gamma_1}$  and  $t_{conf;\gamma_2}$ . Each of the configurations contains the modes extracted from one of the clusters. In the example, the extraction process results in two process modes for cluster  $\gamma_1$  and three modes for  $\gamma_2$  respectively. Now, we can derive an activation function for process  $P_{Var}$  that considers the consumption rates of the individual modes as well as the variant selection. This function is represented as a set of activation rules as follows:

$$\begin{aligned} a_1 &: (C_{In}.num \geq x) \wedge (C_V.num \geq 1) \wedge ('V 1' \in C_V.tag) \mapsto conf_1 \\ a_2 &: (C_{In}.num \geq y) \wedge (C_V.num \geq 1) \wedge ('V 2' \in C_V.tag) \mapsto conf_2 \end{aligned}$$

Note, that  $x$  and  $y$  result from the parameter extraction process. The corresponding activation condition only ensures that there are enough available tokens on the incoming channel  $C_{In}$  to execute the process in the particular mode. However, the actual decision about the configuration and thus the function variant solely depends on the tag of the token on channel  $C_V$ . Clearly, these rules can be further refined to map input token predicates to actual modes by taking into account more information about the associated clusters. Nevertheless, the given rules are sufficient for the representation of function variant selection.

## 5 Examples

In this section, we discuss one theoretical design scenario to demonstrate the advantages of the proposed approach. Finally, a larger example from the video processing domain is presented.

The single SPI graph in the Figure 2 represents two independent systems or applications, each of those can be simply derived by replacing the interface  $\Psi_1$  by either cluster  $\gamma_1$  or cluster  $\gamma_2$ . The advantage of a representation with variants becomes obvious when considering the subsequent synthesis steps module selection, allocation and scheduling. Instead of an independent synthesis cycle for each application, the approach featuring a representation with function variants provides the possibility of overall system optimization and potentially decreases the total design time.

System optimization effort is usually targeted to minimize the (hardware) cost of a system as long as a correct timing behavior can be guaranteed. Considering commonalities between applications during synthesis helps to facilitate reuse of components thus reducing design cost. Independent synthesis of two applications may result in two completely different hardware architectures, even if the systems overlap in large parts of their functionality. The independent synthesis results for the systems of Figure 2 are shown in Table 1 (Application 1 and Application 2). The columns 2 and 4 denote which parts of the design are mapped to software and hardware parts, columns 3 and 5 represent the associated processor and ASIC cost, the total cost is given in column 6 and the design time is shown in column 7. Both systems can be best synthesized by implementing the processes  $P_A$  and  $P_B$  in software and to use ASICs for the cluster  $\gamma_1$  or  $\gamma_2$  respectively.

|               | Software      | Hardware | Total                | Time |    |     |
|---------------|---------------|----------|----------------------|------|----|-----|
| Application 1 | $P_A, P_B$    | 15       | $\gamma_1$           | 19   | 34 | 67  |
| Application 2 | $P_A, P_B$    | 15       | $\gamma_2$           | 23   | 38 | 73  |
| Superposition | $P_A, P_B$    | 15       | $\gamma_1, \gamma_2$ | 42   | 57 | 140 |
| With variants | $\Psi_1, P_B$ | 15       | $P_A$                | 26   | 41 | 118 |

Table 1: System Cost

As mentioned in the introduction, it is often desirable to use a common target architecture (hardware) for a set of applications. Then, the particular application can be fixed by a simple configuration step either by downloading the software part into an EPROM (production variants) or as part of a bootstrap (run-time variants). This enables vendors to delay their decision about the actual application, so they are more flexible and can react to changing market situations quickly. Reconfiguration is an issue, too.

One possible approach to such flexible architectures is to superpose the individual implementations to form the desired architecture. Line 3 (Superposition) of Table 1 shows the resulting system cost for the known example. Since the software processes  $P_A$  and  $P_B$  are part of both applications, the software part can be reused directly from the two independent implementations without additional processor cost. In contrast, the hardware parts are different and both have to be included in the superposition. Clearly, the hardware cost adds up.

The disadvantage of this procedure is that optimization is limited to single applications without considering the final superposition step. The possibility to specify different variants in one single design representation enables overall system optimization on a set of applications as a whole. Such an optimization might result in a different mapping of processes to resources. For the example system of Figure 2, this has the following effects. Now, process  $P_A$

has been moved to hardware while both variants (clusters) of interface  $\Psi_1$  are implemented in software. Since the clusters  $\gamma_1$  and  $\gamma_2$  are mutually exclusive at run-time, the available processor performance is not exceeded. Each of the clusters only needs to share the processor with process  $P_B$ . Apparently, the hardware cost has increased compared to the individual applications (lines 1 and 2 in Table 1). However, the overall benefit results from the decision to map  $P_A$  to hardware. Since  $P_A$  is contained in all variants (applications), the same ASIC can be used for both implementations. In consequence, the hardware cost is remarkably lower than that of the superposition in line 3.

Additionally, a beneficial side effect of a synthesis method using variants over independent synthesis is that the overall design time is expected to decrease. The explanation is straight forward. When synthesizing  $n$  systems individually, a process that occurs in all applications, i. e. that is not variant (or application) dependent, has to be considered  $n$  times. In the proposed approach, such processes need to be considered only once during the synthesis of all applications. This decreases the total number of synthesis decisions to be made. As a result, we expect a shorter overall design time.

As a larger example, we use an industrial reconfigurable video system[3]. It consists of two parts, one process chain that performs the actual computations and one controller process. The chain operates on a video data stream, applies certain functions to the data and outputs the modified stream. Each of the chain processes has an associated set of function variants, and the controller dynamically switches between them in reaction to user requests. To simplify matters, the chain of the example system in Figure 4 consists of only two processes,  $P_1$  and  $P_2$ . Process  $P_{Control}$  models the controller. The additional processes  $P_{In}$  and  $P_{Out}$  serve as valves for the stream. They are necessary to avoid buffer overflows and the output of invalid images that may result from reconfiguring  $P_1$  or  $P_2$ .

At the occurrence of a user request, process  $P_{Control}$  sends certain tokens to  $P_1$  and/or  $P_2$ . These tokens contain information about the selection of a new function variant. Since the following explanations are identical for  $P_1$  and  $P_2$ , only  $P_1$  is discussed. With the concepts introduced in this paper, the modes and the activation function of process  $P_1$  can be extracted from the individual function variants. In this special case, the function variant selection is sensitive to available tokens on the channel  $C_{Req-1}$ . As previously mentioned, these tokens are generated by  $P_{Control}$ . They additionally have an associated tag. Depending on this single tag, the activation function of  $P_1$  selects a particular mode that belongs to the new desired function variant. This consequently results in reconfiguration of  $P_1$ . Thereafter,  $P_1$  generates one token on channel  $C_{Con-1}$  to confirm its completion to  $P_{Control}$ . The generation of this token is not part of the reconfiguration step but part of the selected mode. The same applies to process  $P_2$ . After the reception of confirmations from both processes,  $P_{Control}$  is in its initial state again. To keep state information from one execution to the next,  $P_{Control}$  sends tokens to itself via feedback channel  $C_{CTRL}$ .

As mentioned above, the valves prevent the chain from generating invalid images. This is done as follows. Together with the ‘reconfiguration’-requests to  $P_1$  and  $P_2$ ,  $P_{Control}$  sends ‘suspend’-requests to the processes  $P_{In}$  and  $P_{Out}$ . While  $P_{In}$  simply destroys all input video data in its ‘suspend’ mode,  $P_{Out}$  replaces all chain output data by the last completely modified image. This suspend mode ensures that no invalid images are produced. An image becomes invalid if either  $P_1$  or  $P_2$  or both are reconfigured during processing that image. After the completion of the reconfiguration of  $P_1$  and  $P_2$  is confirmed,  $P_{Control}$  generates a ‘resume’-request token on channel  $C_{In}$ .  $P_{In}$  receives this token and resumes execution in its ‘normal’ mode, that is passing all incoming video data to channel

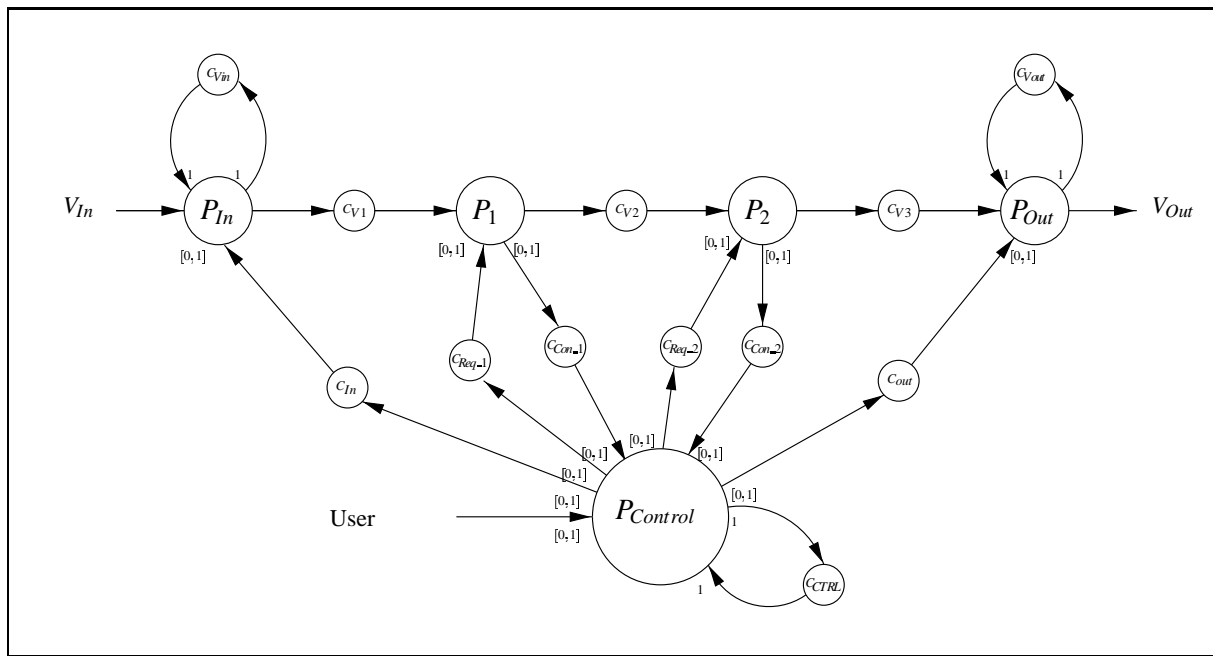


Figure 4: Reconfigurable Video System

$C_{Vj}$ . Additionally, it adds a certain tag to the first image that has passed after resuming. When this tag reaches  $P_{Out}$ , this process resumes its 'normal' mode, too, and the reconfiguration sequence is over.

## 6 Conclusion

We have presented a novel approach for the representation and selection of function variants. The approach considers function variants that are selected before the system operation and remain static during the system's execution (production and run-time variants) as well as function variants that may be selected during run-time by a higher level component (as in reconfigurable architectures). All different types of variants are represented using the same constructs.

In contrast to previous approaches [6, 5], the constructs representing the variants do not degrade the optimization potential for synthesis (no serialization of variants or tasks). Furthermore, the decentralized nature of our approach facilitates the reuse of system parts. Another benefit is the capability to naturally model the process of reconfiguration of system parts.

A set of examples, including an industrial reconfigurable video processing system, have been used to explain and validate our approach.

## References

[1] P. Chou and G. Boriello. An analysis-based approach to composition of distributed embedded systems. In *Proceedings Codes/CASHE '98*, pages 3–7, Seattle, USA, March 1998.

[2] R. Ernst. System architectures. Talk at NATO ASI on System Level Synthesis, August 1998.

[3] R. Ernst, K. Henriss, and P. Rueffer. Software signal processing on image-engine platforms. Technical report, TU Braunschweig, August 1998.

[4] A. Jerraya et al. *Hardware/Software Co-Design: Principles and Practice*, chapter Languages for System-Level Specification and Design. Kluwer Academic Publishers, Boston, USA, October 1997.

[5] A. Kavalade and P. Subrahmanyam. Hardware/software partitioning for multi-function systems. In *Proceedings ICCAD '97*, pages 516–521, San Jose, USA, November 1997.

[6] K. Kim, R. Karri, and M. Potkonjak. Synthesis of application specific programmable processors. In *Proceedings DAC '97*, pages 353–358, Anaheim, USA, June 1997.

[7] Philips Semiconductors. *TriMedia Processor*. <http://www.semiconductors.philips.com/trimedia/>.

[8] D. Ziegenbein, R. Ernst, K. Richter, J. Teich, and L. Thiele. Combining multiple models of computation for scheduling and allocation. In *Proceedings Codes/CASHE '98*, pages 9–13, Seattle, USA, March 1998.

[9] D. Ziegenbein, K. Richter, R. Ernst, J. Teich, and L. Thiele. Representation of process mode correlation for scheduling. In *Proceedings ICCAD '98*, San Jose, USA, November 1998.