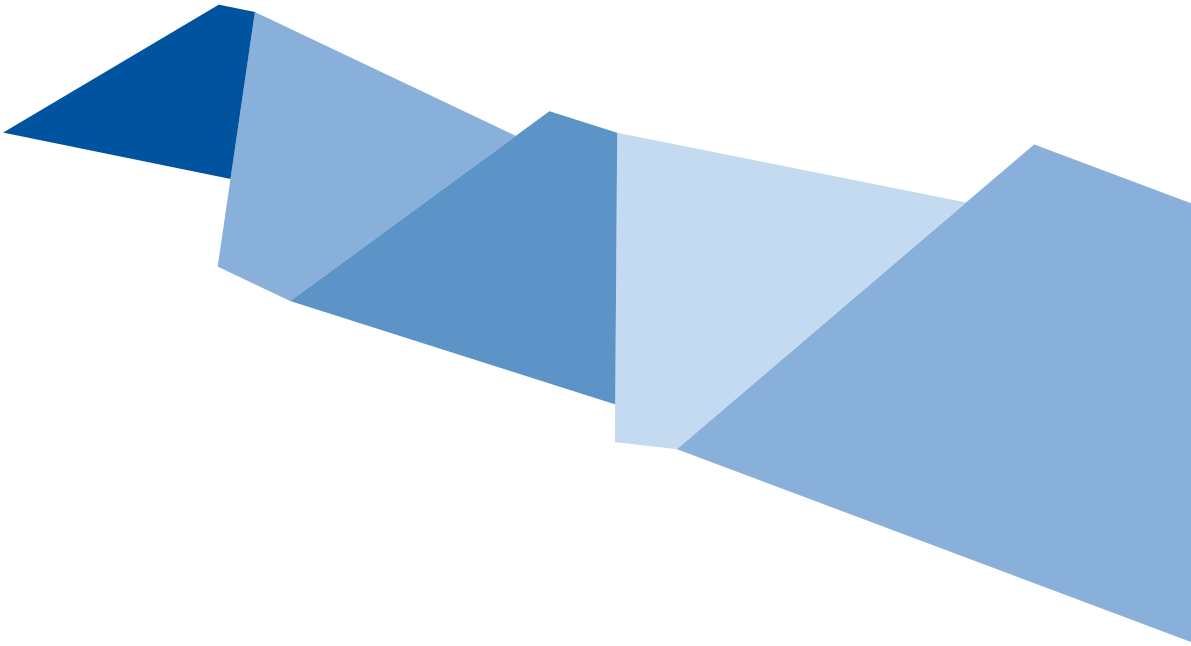


Process Interaction Diagrams are more than Process Chains or Transport Networks

Manfred Nagl

[Nag21d] M. Nagl:
Process Interaction Diagrams are more than Process Chains or Transport Networks.
In: RWTH Aachen University, Technical Report. AIB-2021-05. Feb. 2021.
www.se-rwth.de/publications/

Department of Computer Science

Technical Report

Aachener Informatik-Berichte (AIB) | ISSN 0935-3232 | AIB-2021-05
RWTH Aachen University | Department of Computer Science | February 2021

The publications of the Department of Computer Science of *RWTH Aachen University* are in general accessible through the World Wide Web.

`http://aib.informatik.rwth-aachen.de/`

Process Interaction Diagrams

are more than Process Chains or Transport Networks

Manfred Nagl
Software Engineering
RWTH Aachen University, 52074 Aachen, Germany

Abstract

Process modelling is a broad field of research in different application areas, especially in informatics. Corresponding notations (transport networks, etc.) usually contain sequences, splits and joins of processes. Between processes there are dependencies, which can have quite different semantics. These semantical relations are usually not explicitly expressed.

In this article, we focus on a notation for a process, which has different aspects influencing the process, not only the input. This allows to connect different processes in specific ways, making clear what purpose the connection has, and thereby characterizing different kinds of dependencies between processes. This extended notation we call process interaction diagrams (PIDs). We can express standard interactions of processes more precisely and are also able to express nonstandard interactions.

These diagrams can be applied in different domains, as mechanical engineering, informatics, etc. Interesting and complex interactions can be studied. The notation can be used for different levels of processes, as lifecycle level, project management of development teams, etc. The usual process notations are extended to express interactions like "a process creates a tool to be used in another process".

Keywords: process modelling, dependency relations between subprocesses, different aspects of a process, interaction of different processes, applications in different engineering disciplines

1 Introduction

There is a variety of *notations for processes*, which can be used for quite different kinds and levels of processes, as /AL 16, JB 96, We 99/ to name only a few. They can be *classified* (i) along their main *application* (logistics, business processes, cooperative work in offices, mechanical production, informatics, etc.), (ii) *where* in an overall process they are *used* (knowledge acquisition, pre development, development, preparation for production, maintenance and customer relationship, etc.), (iii) according to their *granularity* (from lifecycle or ERP to fine-grained tasks), or (iv) whether they are *static* or allow *dynamic* changes, and so on.

Process *research* can be *classified* into mining, analysis, formalization, classification, application, evolution etc. The latter can be seen on single process level, on knowledge for processes and their change, on process type level, on knowledge level for specific processes and so on.

In most cases they connect processes mostly along output to input (the output of one process is the input of the next) and, therefore, corresponding to *chain dependencies*: The second process is dependent from the first and can only start, if the first process has delivered a necessary result. Such notations are widely and successfully used.

All these notations can be used to structure processes by composing them from simpler ones (subprocesses) and thereby building nets of these subprocesses by dependency relations. In the graph theoretic sense these nets are *transport networks* /De 74/ which usually have a starting node (source) and an ending node (target). The nodes of the network can be connected to form chains, splits, and joins. For organizational aspects of one specific process this may suffice in many cases.

In this paper we look more carefully on the way, how connections can be organized and denoted. Especially, we look what aspects influence a process, and take these aspects as targets of dependent process edges. Thereby, we *differentiate dependency* relations. We call the corresponding notation *process interaction diagrams*, in short PIDs, thereby adding a further meaning to the abbreviation "PID" /PID 21/. Roughly speaking, these diagrams differentiate incoming edges according to their purpose and, thereby make these networks more semantical, corresponding to the different ways, processes are connected.

Predecessors of PIDs in *simpler forms* are SADT-diagrams /MM 88/, T-diagrams of N. Wirth /Wi 77/ for explaining compiler bootstrapping, or component / connector notations, as /IC 04/.

The *paper is as follows*: After having introduced the different aspects of a process, which influence the way processes can be connected in PIDs in section 2, we discuss quite different examples of process interactions in PIDs in section 3. We present interesting and nontrivial examples of PIDs in section 4. A summary / characterization of the results and the list of references closes this paper.

In this paper we concentrate only on *process* interaction, although – as it was discussed in /HJ 08, NW 94/ – a process is also tightly connected to its resulting *product*, and both also to the *resources* needed (active as actors or passive as pre results), the latter two in abstract form (e.g. abilities for abstract ressources) as well as in actual form (persons with competences).

2 Characterizations of Processes

A *process* is influenced by different aspects, see fig. 1. The process has an *input*, and produces an *output* of quite different kinds, as we are going to learn. The process has a *goal*, has to regard different *constraints*, and follows a certain *way to proceed*, expressed from rather vague to precisely determined. The process has an *actor*, usually for a complicated task a human being or a group of humans. The actor uses *tools* or available *partial results/ solutions* to be a part of the result of the process. Actors apply *knowledge* and / or *experience*. In /NF 03/ even more aspects of a process are discussed. We omit this here to keep the discussion simpler.

The upper part of fig. 1 corresponds to the *planning part* of the process (goal, constraints, way to proceed), the lower part determines the path in *direction* to the *solution* (partial solutions or

tools, actor, and experience/ knowledge). To be more precise, one should distinguish between a task and the process to solve the task. By reasons of simplicity, in this paper a process stands for both. Altogether, we have seven 'input' aspects and only one for the output.

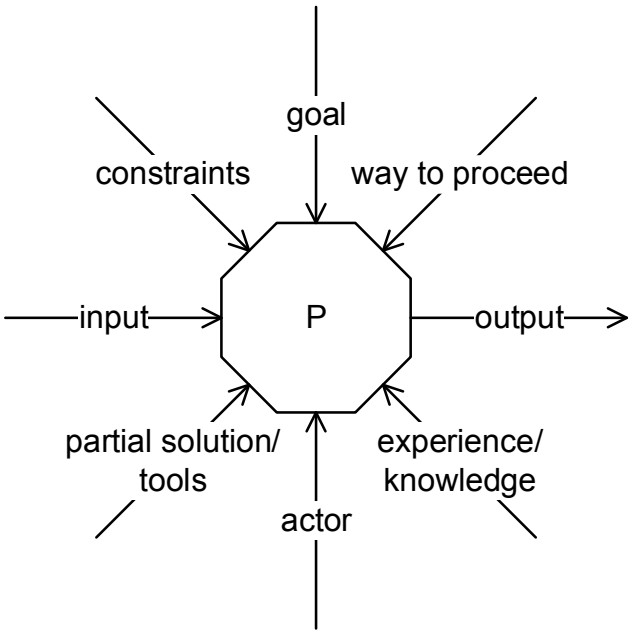


Fig. 1: Different aspects influencing a process `P` and its output

The *actor* of a development process is usually a *human*. In automatic processes the actor can be a *machine* executing a program. In this case, the goal, the constraints, and the way to proceed are or should be incorporated in the automatic program. The same is true for knowledge and experience. The machine (actor) determines the level of the program (preciseness, degree of formality, etc.).

A process corresponds to the solution of a task, which solves an underlying problem. The task may (a) demand for *creativity*, like acquisition or extension of knowledge, pre-development of a product, development of a novel and nontrivial product, extending a solution to a solution for a family, detecting deep reuse for a class of solutions by remarkably changing the process, and alike. In all these cases, the process is not determined, it asks for creativity and new ideas. The task may (b) be rather determined, like often executed processes in business administration or in production of mechanical engineering. Finally, a process may (c) run automatically, as to be found in process automation and control. In this case, everything is running automatically being determined by a software program, eventually including interactions of the operator.

Processes can be found on different *granularity levels*: We find (i) *coarse grained* processes, e.g. on lifecycle level, where we only distinguish subprocesses without looking into their structure, like design, programming. It can be (ii) *medium grained*, where we go down to identify actions of single developers in order to manage these developers, without regarding how a developer does his/her job, like the implementation of a certain component assigned to a developer, or how a subsystem is decomposed and the parts are assigned to different devel-

opers. Finally, processes can be (iii) *fine-grained*, e.g. to look, how a designer is doing the task in detail assigned to him/ her. Even one further level down, we (iv) find the actions of a tool, used by a developer, to facilitate his/her job. Of course, tools can be used on any granularity level of processes.

Furthermore, development processes may be (1) *local* to a certain department of a company, (2) happen inside a company but involving *different departments*, or it may (3) spread over *different companies*. So, we distinguish local, from integrated, and inter-company processes.

What we have said up to now is completely independent from the *level* and *nature* of a *process*. It also applies to any kind of processes, from research, knowledge acquisition, predevelopment, development, realization, preparation, production, to after sales and maintenance. I the last sentence we had mechanical engineering in mind. It could also have been, process engineering, material production, or anything else.

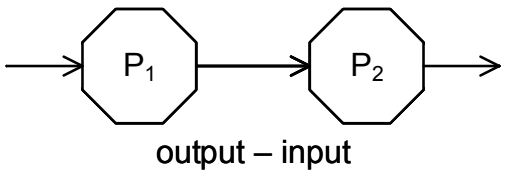
Development *processes* of teams of developers have a certain *structure*, which can be found in every engineering discipline /Na 96/ determining or changing the requirements, the design, and the realization, the latter in engineering often called detail engineering. The *product* of a process is a complex *configuration* of different and mutually dependent artifacts.

3 Process Interactions

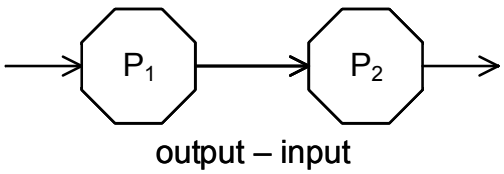
Process Chains for Different Purposes

In fig. 2 we regard usual chains of processes in the sense that one process produces a result which, in the next step, is handled by the following process. However, the *purpose of handling* is *different*. The examples are from the development level. In the first case, the second process P_2 makes the next step of a process decomposed into steps, in the second example it produces a result consistent to the result of P_1, in the third case it formalizes a given non-formal result or it makes a formal result operational. Further examples are possible. The specific purpose of the output/ input relation can be given by an annotation to the relation.

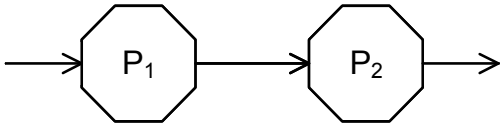
The purpose of these examples is to demonstrate that the *chain dependency relation* between P_1 and P_2 can have *different semantics*. The examples are on the same logical level and they are typical situations occurring in development processes carried out by humans. The examples could also be from any development domain and any application domain (informatics, production engineering, process engineering, etc.).



P_2 makes the next part of a solution.
a) Example: successive computation



P_1 produces a result, which is consistent to the output of P_2.
Example: P_1 delivers requirements for
b) the design of P_2



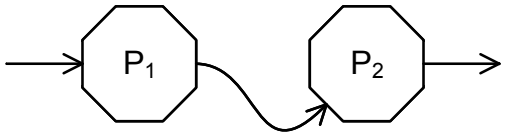
P_1 produces a result which is made
c) formal or precise by P_2

Fig. 2: Process chain relations for different purposes

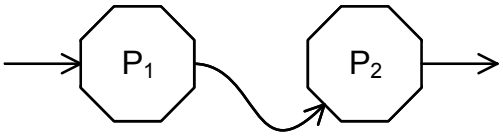
Processes for Creating Components or Tools

(a) Fig. 3.a shows that the process P_1 can deliver a *result*, which makes the *process* P_2 *easier*, as (a) P_1 delivers a partial product, which is helpful for P_2, or (b) P_1 delivers a *tool*, which *helps* for the process P_2 (fig 3.b). Please note that these supports (a) and (b) are possibly not restricted to P_2. If this would be the case, the process P_1 would deliver a part of the solution according to the scheme of fig. 2.a, which then is completed by P_2.

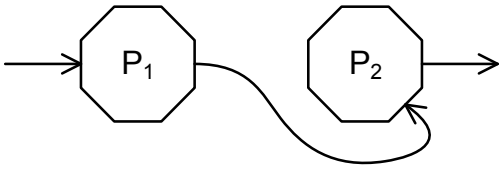
In fig. 3.c, P_1 delivers a result, which supports the actor of P_2, by delivering explicit knowledge or experience, helpful for P_2. This can be e.g. a checklist of items, which helps not to forget an aspect important for the process P_2. Again, annotations can be used.



P_1 creates a result which helps for P_2,
a) e.g. a needed component



P_1 delivers a tool which facilitates the
b) process P_2

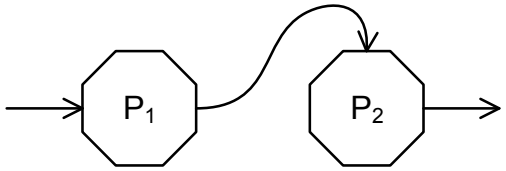


P_1 improves P_2, e.g. by delivering
c) experience/ knowledge in explicit form

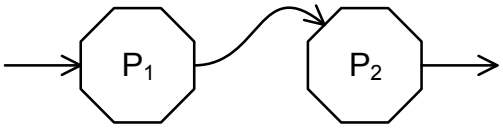
Fig. 3: Actor support by delivering a part of the solution, a tool which helps, or valuable experience/ knowledge

Clarifications for other Processes

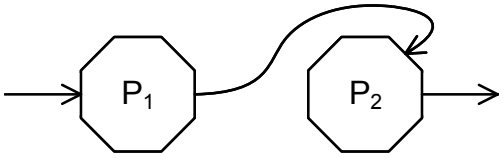
The process P_1 can define/ make precise the goals of P_2 (fig. 4.a), the constraints for P_2 (fig. 4.b), or the way to proceed in P_2 (fig. 4.c). The latter can be from vague and exemplary to formal and complete.



a) P_1 determines/ improves/ clarifies the goal of P_2



b) P_1 determines the constraints for P_2



c) P_1 determines/ improves how P_2 should proceed by giving the steps, a plan how to proceed, or a methodology

Fig. 4: Clarifications: P_1 delivers results for different aspects of P_2

Summing up, we see that the process P1 could deliver a *helpful result* for *any of the 'input' aspects* explained in fig. 1, see figs. 3 and 4. The composition of both processes P_1 and P_2 is different from those discussed in fig. 2. In all cases, we have a dependency of process P_2 from process P_1. However, we recognize that the semantics of the dependency is different for all cases. Annotations may give further distinctions.

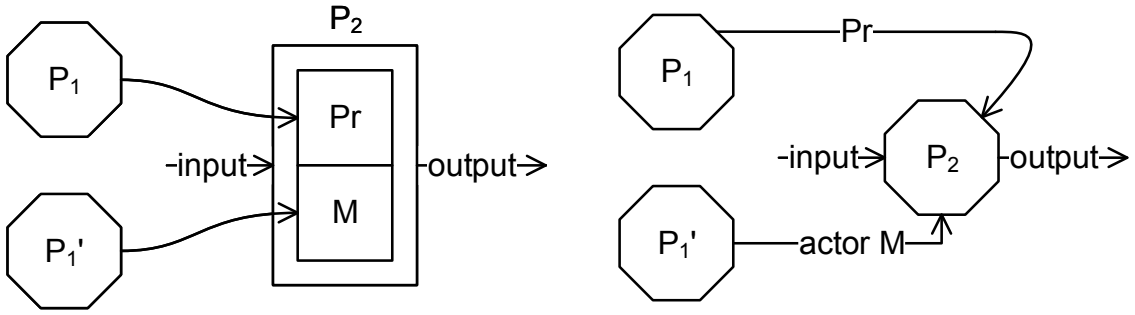
Actors and Machines

In all examples of above, we had *human actors*. A human actor is *intelligent* and *creative*. The actor person can interpret constraints, evaluate a goal and compare a process result with the goal, it can follow advices to proceed, he/ she can build in partial results or apply tools to facilitate or improve a solution. The person can use experience or knowledge to apply. This is true for a scientist who is extending knowledge as well as for a worker, who is carrying out a fabrication subprocess.

What is to recognize, if the *process* is *automatic* and what to obey, if we switch from a human to an automatic process? In informatics terms, an automatic process is a *program* executed by a *machine*. The program may be an executable specification, or a program formulated by an interpreter language together with an interpreter. It can also be a program Pr of a programming language L, together with a machine for this language (compiler, runtime system, and target machine).

Looking on fig. 4 and on P_2, we now discuss that P_2 is an *automatic process*. How does this change the situation? P_1 can create the program and P_1' the machine. Both are human processes. The program with the machine is the automatic process, see fig. 5.a.

A different explanation in our notation is that P_1 creates the program, i.e. the way to proceed, and P_1' the actor of P_2. The program has been written to follow the goal, to regard the restrictions, to use partial solutions or tools, and to adopt knowledge and experience. This, altogether, is incorporated and fixed in the program Pr. So, all the other aspects of P_2 (see fig. 1) are determined by the properties of the program. They are no more important for P_2. Fig. 5.b shows this situation.



a) **b)** in P_2 all other aspects are fixed

Fig. 5: Automatic actors: programs and machines

4 Examples of Process Interaction Diagrams

Production Engineering

We now switch to the field of *mechanical engineering*. That differs from software development in two ways. (a) The products are produced after their development in a more or less complicated production process. Furthermore, (b) also the facilities for production have to be looked at. Usually, this production machinery is used for different products. In the case of a novel product it might be that the machinery has to be developed specifically.

Both is usually not the case for *software*. Only in *rare cases*, there is something like a *production* of software. Even in these cases, the production is mostly only configuration and delivery.

We now discuss the *interaction* of these three *processes* development, production and use, and production facility development /NF 03/. The production of the facility is usually done in other companies and these machines are in most cases produced to serve for different production processes. The three processes are in *three different dimensions*, see fig 6.

The example is on the level of *coarse processes* (level of lifecycle), which means that the processes are not structured internally. For example, the *product development* is usually done in steps belonging to *different levels* (requirements, conceptual and detailed design, detail engineering), which is not done in fig. 6. These levels are structured internally to express that different people do different things, which have to be consistent to each other. This we call middle-grained or *organizational level*, because it structures, how the cooperation of different developers is organized.

In fig. 6 we see three process *chains*, which are *orthogonal* to each other:
(a) In the middle and drawn horizontally, there is the *physical product lifecycle*. A product is produced, later it is used and maintained (physical maintenance), and even later we see the recycling. The production process uses parts, which are manufactured by suppliers. The chain has the usual output/ input dependency relation.
(b) On top and drawn down, we find the *development lifecycle*, again presented in a coarse form with the usual output/ input relation. We start with pre-development, to show that the development is possible and can result in a reasonable product (we build a prototype and decide to go further or to stop). Then, we develop the product, using different internal steps. If that was successful, we start with the production preparation, also consisting of different internal steps. The results delivers know-how, which either is useful and necessary for the manufacturing process or which delivers corresponding experience/ knowledge. Production preparation also delivers constraints and ways to proceed (e.g. NC programs).
(c) At the bottom and drawn upwards in fig. 6, we find the process to *provide* the *platform for manufacturing*. The platform is developed in two parallel steps: (i) How the production platform is built up (determination of corresponding machine tools and how they are configured), and (ii) developing the corresponding automation and control programs. They are both combined in the manufacturing *plant*. Typically, the manufacturing machine tools are not specifically developed and built up (that can happen in specific contexts, where no corresponding machine tools are available on the market). However, the machine tools may need some adaptation. Also, the automation and control infrastructure is available, but specific programs have to be written, adapted, or generated. The production plant together with the corresponding personnel is the actor of the manufacturing process.

Summing up, we see that fig. 6 consists of three coarse *processes* in *three dimensions*: physical production, development of the product to be later produced, and making the production infrastructure (plant) available. The processes are connected with the *business administration process* (ERP) for sales, production of a number of products, logistics , maintenance of products in a repair center, after sales relations, etc. This ERP process is not shown in fig. 6.

Any of these four processes is a cross-company process. The used intermediate products of all these processes also have a development and need a manufacturing infrastructure. All processes *interact* in interesting and *different ways*. This is the main argument to sketch the situation here.

Now, we make the process interaction even more complex by introducing further new tools which come up in the above three dimensions, see again fig. 6. Firstly, (i) a new *tool* is introduced which makes the *product development* more efficient and/ or helps to avoid mistakes or errors. Secondly, (ii) a mechanical tool is developed needed for the *specific production* process. Thirdly, (iii) a tool is introduced which helps in the *design* of the *automation and control* part, again for efficiency and quality improvement. These tools introduce further dimensions.

The new tool for development (i) influences and changes the development process. Especially, it also *changes* the *experience* / *knowledge* of this process and it also changes the *actors* of this process. The change might be gradual or dramatic. The same is true (iii) for the new tool helping to develop the automation and control part. A tool helping for configuring the produc-

tion plant (not shown in fig. 6) would influence the configuration process in the same way. Another role has (ii) the tool for a tool machine in the production process. It is produced (usually without a development or a complicated production process), and it has to be replaced after some time, due to wear and tear.

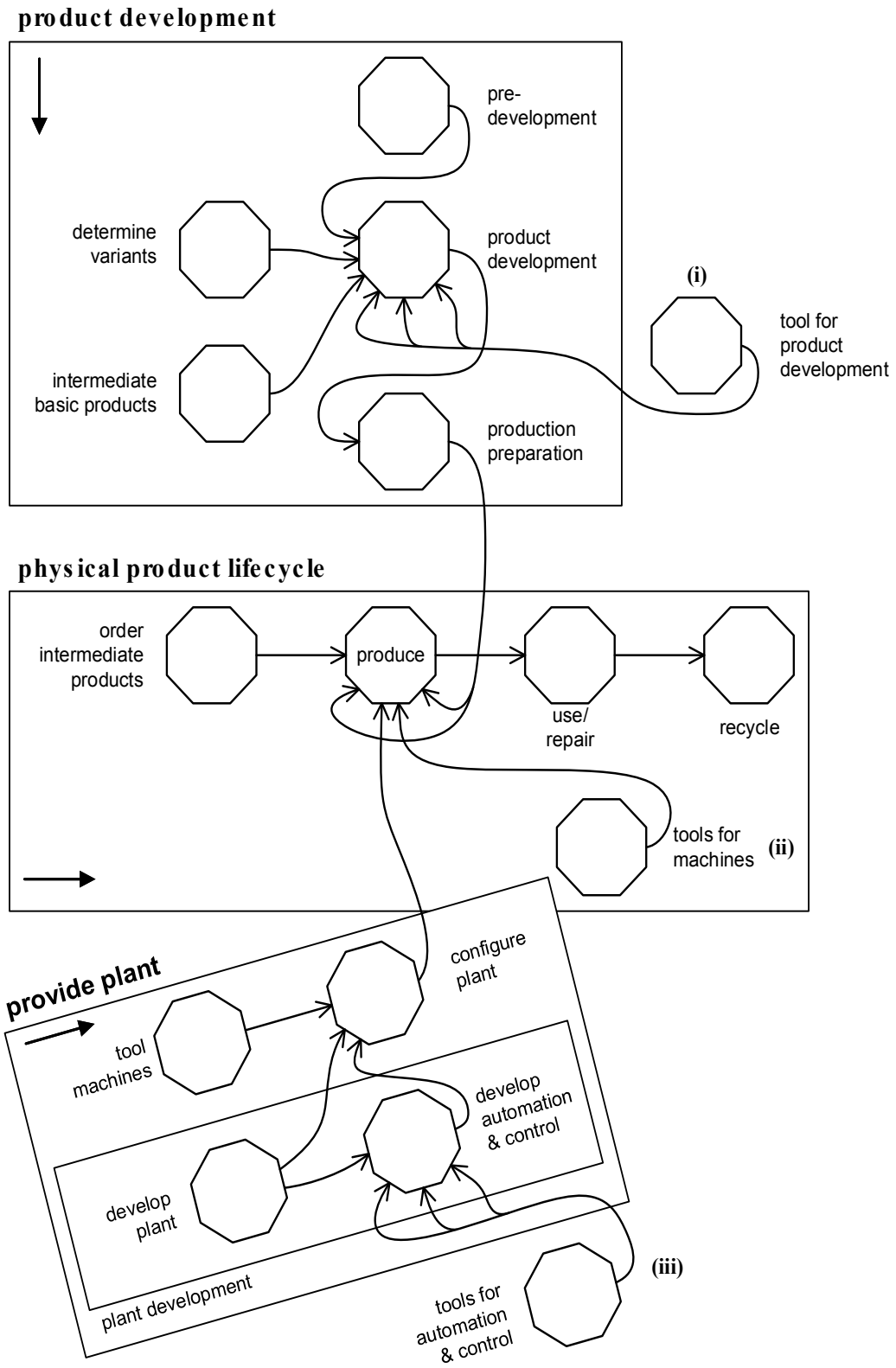


Fig. 6: Interleaved and orthogonal processes in production engineering

Mostly tool development is *not done in parallel* to product or facility development or production. Usually it is used for the next product cycle. However, it can be that it is even needed *in in the running process*. The development of a new production tool is usually done in parallel.

The 'Analogous' Situation in Software Engineering

Now we switch to a similar situation as that of fig. 6, but now in the field of *software development*. The figure changes noticeably. This change characterizes the difference to a field, which produces mainly physical products (machines, chemical plants, etc.) on the one hand from a field, where mainly nonphysical products (software) are created. So, not only the *characterization* of the *change* but also the *difference* of the *fields* are interesting.

As in software development immaterial products are created, there is usually *no production process* (horizontal chain of fig. 6). Software is usually only copied and distributed. I only saw once something like a production process for software products. That was with the company DATEV, which provides and delivers software and IT services for tax consultants /DA 21/. As there are 40.000 consultants being supported by DATEV, which have quite different profiles and also various and different IT infrastructures, delivering an update, extension, or porting of software is a rather complex configuration but also delivering process. In such a situation, we find something like a 'production' process. That, however, is not generally typical in the case of software.

As there is usually no production process, also *production preparation* - the last big step of product development in mechanical engineering - does *not appear*, or it appears only in a primitive form. The same is true for the infrastructure to produce the products in production engineering. The production is immaterial, so does *not need* to be *produced by a sophisticated infrastructure*.

Therefore, mainly the *product development* process of fig. 6 *remains*. This process, however, can include all the relations between subprocesses, we have sketched in this paper. Furthermore, software as an immaterial product, is more flexible w.r.t. changes and also transformations in direction of applying reuse strategies.

Hierarchies in and Management of Software Development

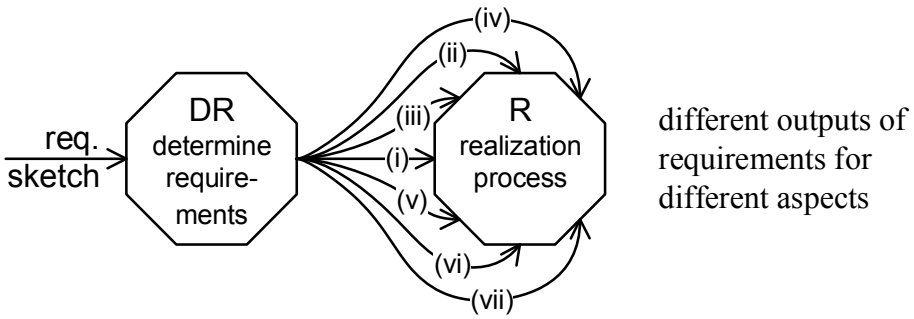
We show some process interactions, which are possible within software development, by sketching some of many situations in this subsection.

Firstly, we look on the dependency relations between *Requirements Engineering*, where the requirements for a future software system or the requirements for a change of an existing software system are made precise and the following *realization* of the system, see fig. 7.a. In this activity area requirements engineering (modelling and changing requirements) quite different results are produced.

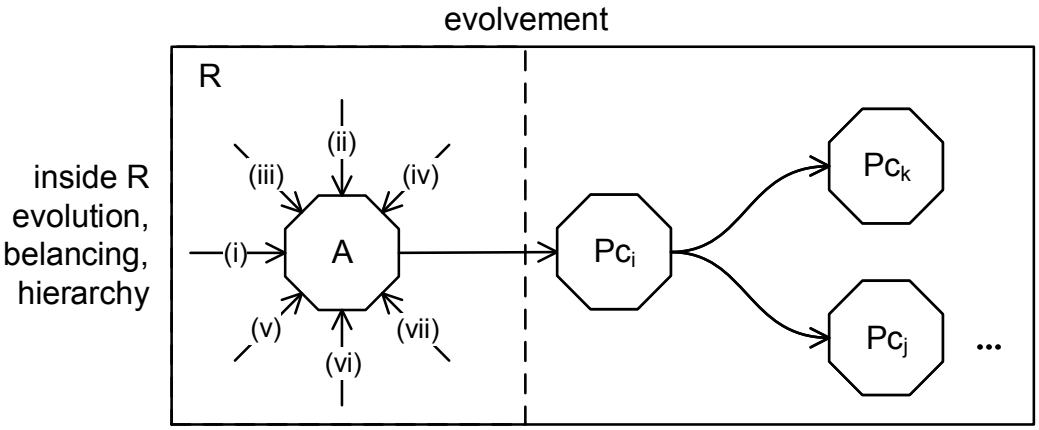
One result is the requirements specification, which determines what the following system is going to do, which is often called *functional requirements specification*. This part is (i) the *input* of the realization process. It is used to build up a system in several steps, which is consistent with the functional specification, see fig. 2.b. The requirements area is producing *fur-*

ther determinations: (ii) the goal of the system to be built or changed, (iii) the constraints which the future system has to follow, e.g. efficiency parameters, (iv) fixing some of the ways to produce the result, e.g. such that the process shares similarities with other development processes already finished or being in parallel progress (e.g. ways to structure the interactive input, such that it is uniform to existing systems). Furthermore, it can (v) determine (v.a) some of the external components to be used or (v.b) some tools to be applied, and (vi) predetermine the development process, e.g. by fixing the corresponding quality assurance procedures. Finally, they may (vii) state which experiences or which knowledge are to be applied.

We see that the requirements area produces quite *different results important* for the following development process which specify what to do (i, ii), the determinations and constraints (iii, iv), determining the product (iv, v), and the process (v, vi, vii). This can be nicely separated and expressed by the *different facets* of process P and A (figs. 1 and 7.a). This example is one of the rare cases, that *all forms of the dependency relations* occur between two processes, here between DR and R of fig. 7.a.



a)



b)

Fig. 7: Hierarchy, balancing, and evolution

The following realization process is *structured internally*, see fig. 7.b. It starts with the process for *architecture modelling* A. Most of the realization process' incoming edges to R have to be directed towards this process (by so-called balancing). This is easy to see, as requirements determinations mostly deal with or influence the architectural structure of the system. Only some of the relations may also influence following component realization processes.

We assume that the architectural process is simple, it can be carried out by one designer and, therefore, need not be decomposed. For the following realization processes for components of

the system we can only say that they appear. But we do not know how many such processes will come up, so we do not know which components have to be realized. This is fixed not until the architectural process A has delivered its result, namely a part or the complete architecture of the system. So, in the *running process* R, by the result of a process A it is determined, how the *following processes look like*, see again fig. 7.b. Furthermore, the dependency relations between these processes are also determined (the process for the component P_{Cj} is dependent on that for P_{Ci}). Thus, a subnet of the process net is determined by the result of A, see again fig. 7.b. This situation we called *dynamics* due to *evolution* /Kr 98, Sc 02/.

It should be noted that the extension of the process net of R to get the processes for the components is done by another process, which does not appear in fig. 7.b. This *process* is usually done by a human, the *process manager* or chief designer, who looks at the architecture (the product of A) and then makes the extension, possibly supported by a tool. The extension in fig. 7.b - a net of subprocesses and their dependency - is the result of this activity. There are *further* and more complex *dynamic situations* possible, some of which we sketch in the summary.

Bootstrapping and Compilers

We come back to the discussion of automatic processes at the end of section 3. We are going to discuss combinations of those automatic processes. The example we take is *bootstrapping* of compilers /Wi 77/. With this example, we show that our notation allows to smoothly *connect human and automatic* processes.

Bootstrapping uses so-called T-diagrams /Wi 77/, see fig. 8.a. A T-diagram denotes a specific automated process, namely a *translation* of programs written in a programming language. The left part of fig. 8.a shows a T-diagram translating programs of a language So (for source) to programs of Ta (for target), written in the language I (for implementation). So, the T form ${}^{So}{}_{M}{}^{Ta}$ is *executable*, if a corresponding machine M for the language M is available (right part of fig. 8.a). This M-program can be executed and translates a So to a Ta program. The program M and the corresponding machine M form a process, see explanation at the end of section 3.

Such T-diagrams can be connected, see lower part of fig. 8.a. If a ${}^{I}{}_{M}{}^{M}$ is available, it is a M program translating I to M, it can be used for the ${}^{S}{}_{I}{}^{T}$ as input, which is an I program. The result is ${}^{S}{}_{M}{}^{T}$, an M program translating S to T. Such combinations can be used for bootstrapping, .i.e. for compiling compilers with reduced effort.

Bootstrapping can be used for different tasks. We discuss here the *extension* of a *programming language* and developing the *compiler* for the extended language, see fig. 8.b. For that we need a compiler for S written in S (e.g. a Pascal compiler written in Pascal /Wi 77/) and one S compiler producing M and written in M, both in green. In two development steps and two automatic steps we get the result.

We take the given ${}^{S}{}_{S}{}^{M}$ compiler and *modify* it (1) to become an ${}^{S'}{}_{S}{}^{M}$ compiler (human process). We *translate* this compiler with the available ${}^{S}{}_{M}{}^{M}$ compiler and get an ${}^{S'}{}_{M}{}^{M}$ compiler (automat-

ic task (2)). This compiler is used in the last automatic step. Then, we (3) *modify* the $^{S'}{}_{S}{}^{M}$ compiler to a $^{S'}{}_{S'}{}^{M}$ compiler (human task), i.e. by using the extended features of S' in the compiler. Finally, we *translate* this compiler, using the $^{S'}{}_{M}{}^{M}$ compiler of step (2) and get the result (automatic process (4)).

Altogether we have extended the $^{S}{}_{M}{}^{M}$ compiler to a $^{S'}{}_{M}{}^{M}$ compiler. Step (2) and step (4) produced an $^{S'}{}_{M}{}^{M}$ compiler. The advantage of the last one of step (4) is that it *makes use of the advanced features of* S' in its implementation. These steps for extension (1) to (4) can be used repeatedly to further extend the language and the compiler.

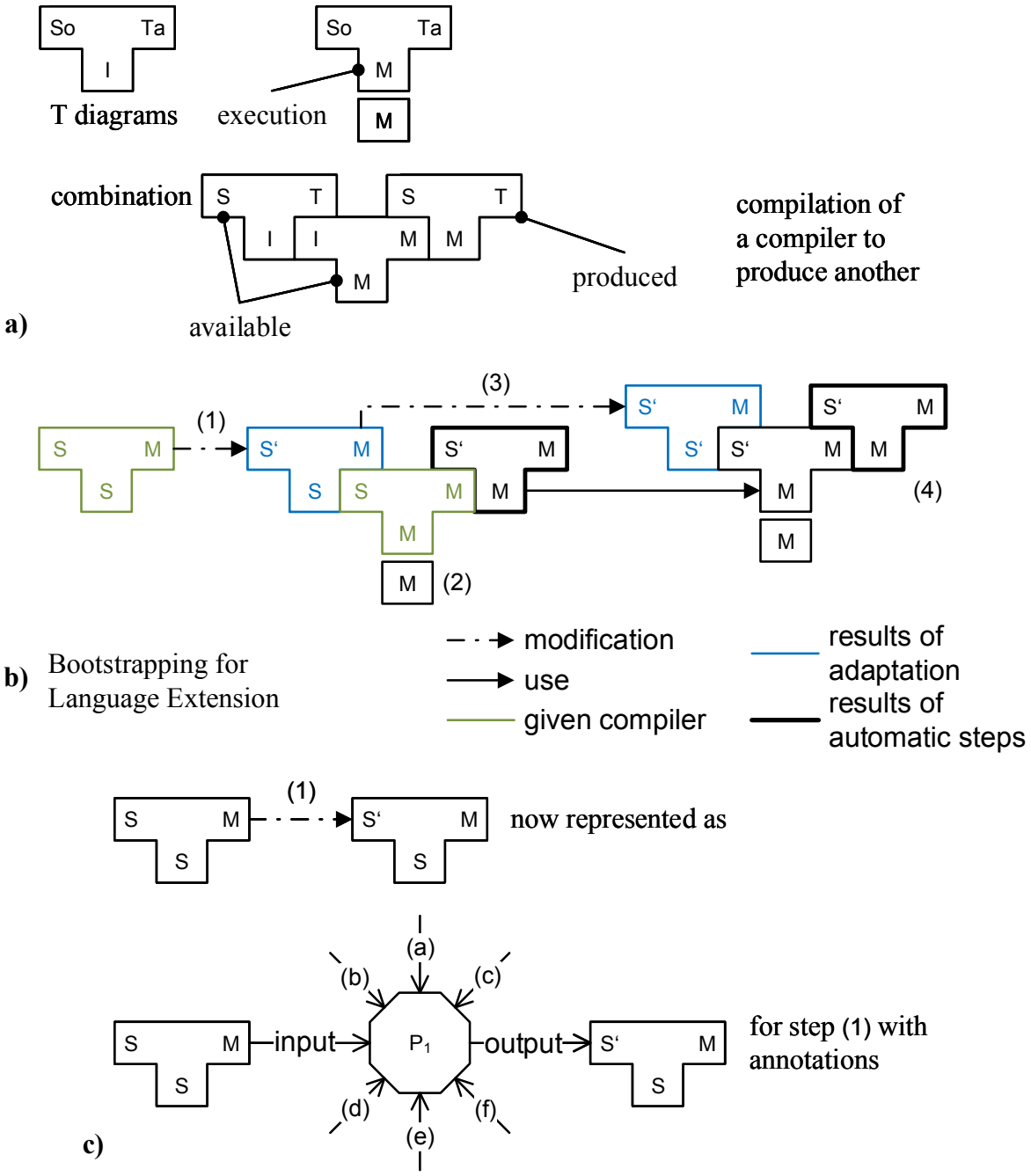


Fig. 8: Bootstrapping for language extension, the PID contains human and automatic processes

The notation of fig. 8.b has *automatic steps* (2) and (4) made precise by combinations of T-diagrams, but also informal *human tasks* (1) and (3), characterized by bold lines.

What has this discussion to do with this paper? We explain this in two steps.

T-diagrams represent automated and very specific processes. They make a translation of two formal and executable languages. And, they are formulated in a formal and executable language. So, the input and output of fig. 1 is formal and fixed and the program how to do is formal and fixed, too. Restrictions, goals, support, knowledge and experience (see fig. 1) are now irrelevant, they all are expressed in the translating program. For this *special case*, we can use a *special notation*, which is done by T-diagrams. We can say, the T-diagram part of fig. 8.a is only an abbreviation for a part usually expressed by PIDs. The expressiveness of PIDs is not needed for this special case. Furthermore, these T-diagrams have the properties of combination and automatic translation.

It remains to show, that the *human steps* are *better explained* by our notation. We look at step (1) of figs. 8.b, 8.c and give the version in our notation for the process P_1, see fig. 8.c. We discuss the different aspects of the process P_1:
(a) The goal is to extend the compiler such that it translates the extended language S'. So, the difference S'-S has to be translated in addition.
(b) The restriction is to take the ${}^{S}{}_{S}M$ compiler and add the compilation of S'-S.
(c) The way to proceed is to directly add the missing source code for the difference, written in the language S.
(d) The support is to start with the given ${}^{S}{}_{S}M$ compiler. The missing part can be directly coded. Or available and more efficient techniques from compiler construction can be used to facilitate coding.
(e) The actor is a person familiar with compiler writing.
(f) Necessary knowledge and experience is bootstrapping, compiler writing (eventually advanced techniques), experience with the programming languages S and S'.

The explanations (a) to (f) may appear as *annotations* in the diagram. You get the corresponding text, if you click an annotation.

The second human step can be described similarly. So, altogether we have a notation consisting of T-diagrams, which we can regard as shortcut abbreviations of the PIDs of this section. They denote special cases. The human processes can be expressed by PIDs with annotations. So, the complete diagram is a PID.

All examples of this section (see figs. 6, 7, and 8) were *nontrivial* and unconventional for process diagrams. Therefore, they demanded for a *suitable notation*. Usual process notations do not offer the necessary concepts.

Another interesting example for showing interaction of processes would have been to show, how a *software development process* and *product* is changed, if we use applications of more or less deep *reuse mechanisms*. Here, the different interactions of subprocesses are interesting for any level of reuse. Especially, it is interesting to look on the change of the development

process when the next level of reuse is applied and how the interaction of subprocesses changes correspondingly. We leave this discussion to a paper on reuse /Na 21/.

5 Summary

What we have achieved

We have generalized process notations in order to explicitly express different *influencing parameters* for processes, not only input and output, but also goal, restrictions, how to proceed, actor, helpful means (directly used components or tools), and experience or knowledge, see fig. 1. The purpose was to make the *dependency relations* between processes - one needs the 'results' of one for the other - more specific.

That allows to express the *interactions of processes* more *precisely*, especially the dependency relations. That is why we named that notation process interaction diagrams (PIDs). We see that the output of one process can have quite different semantical influences on another process, not only input, but also to define the goals, the restrictions, the planned proceeding to go, the actor, components or tools, experience and knowledge for another process, see figures of section 2. These semantical dependency relations make the diagrams more *meaningful*.

In section 3, we discussed some *nontrivial examples*, as interleaving processes (design of the product and planning its production, life cycle of the product, and design, and configuration of the producing infrastructure. A similar example of a simpler complexity was discussed for software engineering. Further examples were the relation between requirements determination and realization, changes due to process evolution, and showing that management and technical processes are intertwined. Finally, the bootstrapping example showed that processes by humans and automatic ones can also interact. All examples were nontrivial and, therefore, have profited from the extended notation introduced in this paper. Section 4 also shows that the ideas presented in this paper are applicable to any engineering domain.

Example processes can be on different levels: coarse (figs. 6, 7.a), middle (7.b, 8), or even fine-grained (not used in this paper). We saw human and automatic processes, different situations of mechanical engineering, application to software engineering, and also software engineering in the systems development domain, namely compiler compilers. So, the notation can be used in quite *different situations* w.r.t. granularity, domains, character of subprocesses, etc.

As in above discussions the output of a process is connected to one of the aspects of another process by a dependency . We could also have combined *processes by gluing* them at the right side of the octagon of fig.1. The upside side is given in bold. Fig. 9 shows the results for the figures 2, 3, 4, and 5 in that glue notation.

This notation can be used only in *simple cases*, as in complex situations one looses track of which side is connected to which side. Furthermore, the order of connections and executions

cannot be seen easily. As an example, combine fig. 9.e with 9.f.

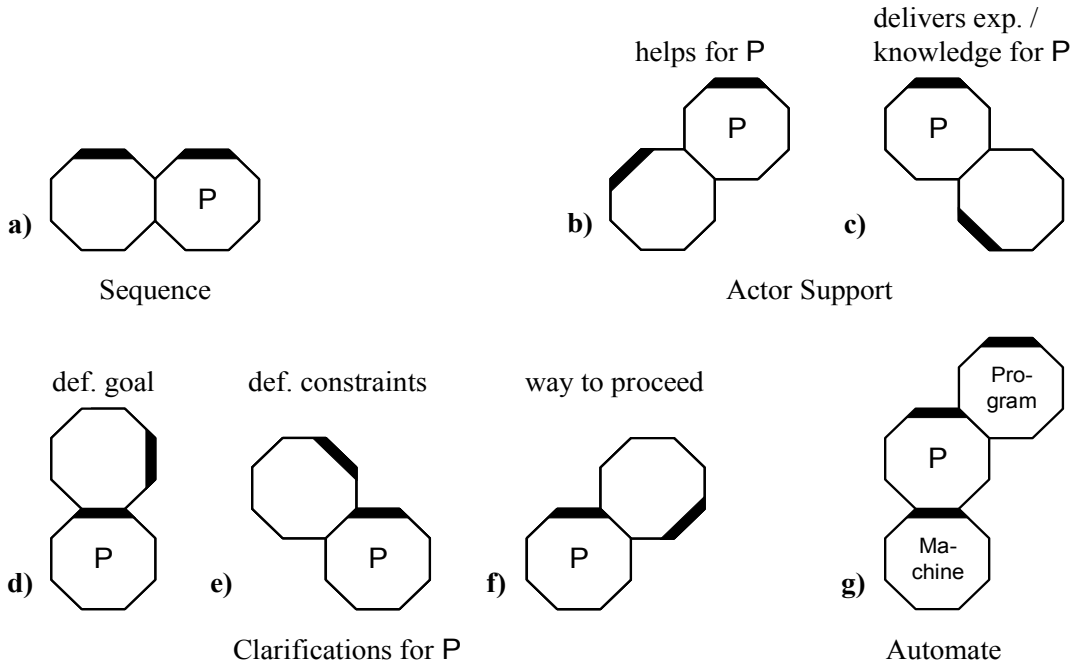


Fig. 9: Gluing process elements, a notation only for simple cases

As we have seen above, the T-diagrams are a *specific notation for a special case*. The same is true for SADT-, SA- and similar diagrams. They are only able to express *some of the aspects* of PID-diagrams (see fig.1).

Activities in Process Research

Some words for our *activities in process modelling*: We have been active for quite a while in that area and mostly there on the middle-grained level, namely how to organize processes, where their parts are carried out by different persons and also mostly in the development field /We 99, NW 94/. There, we mostly specialized on the models and tools aspects. Especially, we focused on *dynamics* problems, i.e. changes that occur in a process in execution. The different aspects are changes due to *evolution* (only in the running process we see by a result, how to proceed further /Kr 98/, see fig. 7.), *backtracking* (a mistake in the development implies that we have to go back in the process; however, we want to preserve useful results and minimize the modifications of others), *extending process knowledge* (new experience and knowledge in form of new process type definitions or workflow pieces should be used in the running process /He 11, Sc 02/), and *cross-company processes* (where all the above problems occur and a grey-box model is needed, which allows cooperation (both sides know of each other) but also protection (of the knowledge of subcontractors) /He 08, Jä 03/. Thus, a central challenge in all these cases was to guarantee *changeability* at *process runtime*.

There were many applications and studies. Those with the most influence were in the IPSEN project dealing with new and tightly integrated tools for *software* development /Na 96/, in the SUKITS project on a posteriori extensions of tools in *mechanical engineering* /NW 99/, and in the IMPROVE project on new tools and a posteriori extensions of given tools in *chemical*

engineering /NM 08/. Most of our studies developed novel tools, but we also studied extensions of given tools /He 11, Wö 10/.

As indicated above, we concentrated in our research on the middle-grained processes (and a bit on the coarse-grained processes, fig. 6, 7.a). Furthermore, we concentrated on the product part (the outcome) of processes, by building tools to make the product of a process easier to get or of a better quality and, thereby, supporting the process as well. We did not regard *fine-grained processes*, i.e. how a single developer is planning or doing his/her work. This was studies in a neighboring group, where activity patterns on this level were retrieved and used to build corresponding *tool actions* on fine-grained level /MJW 08/.

As indicated above, we specialized in this paper only on the process part of process modelling and ignored the product, the human actor, and the support part, all of them on abstract and detailed level. The research we carried out (see citations of above) included these parts. The results, presented above in this paper, could be used to reformulate and *extend* the *process* part of the literature cited above. Therefore, this paper remains in the tradition of having a clear *focus* in this broad domain of process investigations.

6 References

/Al 16/ W. van der Aalst: Process Mining – Data Science in Action, 2nd ed., Springer 477 pp. (2016).

/ /DA 21/ DATEV. https://www.datev.de/web/de/m/ueber-datev/dasunternehmen/?stat_Mparam=int_footer_azg_das-unternehmen, access Jan. 2021

/De 74/ N. Deo: Graph Theory with Applications to Engineering and Computer Science, 467 pp., Dover Publications (1974)

/He 08/ M. Heller: Dezentralisiertes, sichtenbasiertes Management übergreifender Entwicklungsprozesse, Doct. Dissertation, 501 pp., RWTH Aachen (2008)

/He 11/ Th. Heer: Controlling Development Processes, Doct. Dissertation RWTH Aachen, 430 pp., Aachener Informatik-Berichte SE 10 (2011)

/HJ 08/ M. Heller/ D. Jäger/ C.-A. Krapp/ M. Nagl/ A. Schleicher/ B. Westfechtel: An Adaptive and Reactive Management System for Project Coordination, Lecture Notes in Computer Science 4970, 300-366 (2008)

/IC 04/ J. Ivers/ P. Clements/ D. Garlan/ R. Nord/ B. Schmerl/ J.R. Ovieda Siva: Documenting Component and Connector Views with UML 2.0, Techn. Rep. CMU-SEI-2004-TR-008

/Jä 03/ D. Jäger: Unterstützung übergreifender Kooperation in komplexen Entwicklungsprozessen, Doct. Dissertation RWTH Aachen, 260 S., Aachener Berichte zur Informatik 34 (2003)

/JB 96/ S. Jablonski/ C. Bussler: Workflow Management Modeling Concepts, Architecture and Implementation, International Thomson Computer Press (1996)

/Kr 98/ K. A. Krapp: An Adaptable Environment for Management of Development Processes, Doct. Dissertation. RWTH Aachen, 196 pp., Aachener Berichte zur Informatik 22 (1998)

/MJW 08/ M. Miatidis/ M. Jarke/ K. Weidenhaupt: Using Developers's Experience in Cooperative Design Processes, Lecture Notes in Computer Science 4970, 185-223 (2008)

/MM 88/ D. A. Marca, D.A./ C.L. McGowan: SADT: structured analysis and design technique, McGraw-Hill (1988)

/Na 96/ M. Nagl (Ed.): Building Tightly Integrated Software Development Environments - The IPSEN Project, Lecture Notes in Computer Science 1170, 709 S., Springer (1996)

/Na 21/ M. Nagl: Shallow and Deep Reuse, Techn. Rep. AIB 2021-6, 13 pp. Dept. of Computer Science, RWTH Aachen University

/NF 03/ M. Nagl/ O.B. Faneye: Gemeinsamkeiten und Unterschiede von Entwicklungsprozessen in verschiedenen Ingenieurdisziplinen, in M. Nagl/ B. Westfechte (Eds.): Modelle, Werkzeuge und Infrastrukturen zur Unterstützung von Entwicklungsprozessen, 311-324, Wiley-VCH (2003)

/NM 08/ M. Nagl/W. Marquardt (Eds.): Collaborative and Distributed Chemical Engineering: From Understanding to Substantial Design Process Support – Results of the IMPROVE Project, Lecture Notes in Computer Science 4970, 851 pp., Springer (2008)

/NW 94/ M. Nagl/ B. Westfechtel: A Universal Component for the Administration in Distributed and Integrated Development Environments, Technical Report Aachener Informatik-Berichte 94-8, 70 pp. (1994)

/NW 99/ M. Nagl/B. Westfechtel (Hrsg.): Integration von Entwicklungssystemen in Ingenieuranwendungen - Substantielle Verbesserung der Entwicklungsprozesse, 440 S., Springer (1999)

/PID 21/ https://en.wikipedia.org/wiki/PID, access Jan. 2021

/Sc 02/ A. Schleicher: Roundtrip Process Evolution Support in a Wide Spectrum Process Management System, Doct. Dissertation, RWTH Aachen, 310 pp., Deutscher Universitätsverlag (2002)

/We 99/ B. Westfechtel: Models and Tools for Managing Development Processes, Habilitation Thesis, 418 pp., Lect. Notes in Computer Science 1646 (1999)

/Wi 77/ N. Wirth: Compilerbau, Teubner Studienbücher Informatik (1977), for bootstrapping see pp. 79 – 84

/Wö 10/ R. Wörzberger: Management dynamischer Geschäftsprozesse auf Basis statischer Prozessmanagementsysteme, Doct. Dissertation, RWTH Aachen, 304 pp., Aachener Informatik-Berichte SE 2 (2010)

Prof. Dr.-Ing. Dr. h.c. Manfred Nagl, Emeritus
Informatics Department, RWTH Aachen University
nagl@cs.rwth-aachen.de

Aachener Informatik-Berichte

This list contains all technical reports published during the past three years. A complete list of (more than 570) reports dating back to 1987 is available from

`http://aib.informatik.rwth-aachen.de/`

or can be downloaded directly via

`http://aib.informatik.rwth-aachen.de/tex-files/berichte.pdf`

To obtain copies please consult the above URL or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen,
Email: `biblio@informatik.rwth-aachen.de`

2018-02 Jens Deussen, Viktor Mosenkis, and Uwe Naumann: Ansatz zur variantenreichen und modellbasierten Entwicklung von eingebetteten Systemen unter Berücksichtigung regelungs- und softwaretechnischer Anforderungen

2018-03 Igor Kalkov: A Real-time Capable, Open-Source-based Platform for Off-the-Shelf Embedded Devices

2018-04 Andreas Ganser: Operation-Based Model Recommenders

2018-05 Matthias Terber: Real-World Deployment and Evaluation of Synchronous Programming in Reactive Embedded Systems

2018-06 Christian Hensel: The Probabilistic Model Checker Storm - Symbolic Methods for Probabilistic Model Checking

2019-02 Tim Felix Lange: IC3 Software Model Checking

2019-03 Sebastian Patrick Grobosch: Formale Methoden für die Entwicklung von eingebetteter Software in kleinen und mittleren Unternehmen

2019-05 Florian Göbe: Runtime Supervision of PLC Programs Using Discrete-Event Systems

2020-02 Jens Christoph Bürger, Hendrik Kausch, Deni Raco, Jan Oliver Ringert, Bernhard Rumpe, Sebastian Stüber, and Marc Wiartalla: Towards an Isabelle Theory for distributed, interactive systems - the untimed case

2020-03 John F. Schommer: Adaptierung des Zeitverhaltens nicht-echtzeitfähiger Software für den Einsatz in zeitheterogenen Netzwerken

2020-04 Gereon Kremer: Cylindrical Algebraic Decomposition for Nonlinear Arithmetic Problems

2020-05 Imke Drave, Oliver Kautz, Judith Michael, and Bernhard Rumpe: Pre-Study on the Usefulness of Difference Operators for Modeling Languages in Software Development

2021-01 Mathias Obster: Unterstützung der SPS-Programmierung durch Statische Analyse während der Programmeingabe

2021-02 Manfred Nagl: An Integrative Approach for Software Architectures

2021-03 Manfred Nagl: Sequences of Software Architectures

2021-04 Manfred Nagl: Embedded Systems: Simple Rules to Improve Adaptability

2021-05 Manfred Nagl: Process Interaction Diagrams are more than Process Chains or Transport Networks

2021-06 Manfred Nagl: Characterization of Shallow and Deep Reuse