Components as Resources and Cooperative Action

Hans-Gert Gräbe

InfAI, Leipzig University, Leipzig, Germany
graebe@infai.org

Abstract. The concept of systems and problem solving are central to TRIZ. Solving a problem along TRIZ ends in a certain *conditional mind game* that requires implementation and operation to "change the world" along the proposed solution. In [5] the question is examined more closely how real-world problem solving and resource exploitation emerge from such a "conditional mind game". Different to resource concepts widespread in TRIZ it turns out that in modern high-tech worlds, qualitative and quantitative determinacy as *specification* is becoming increasingly important for the selection, search and preparation of resources to provide *couplings* between systems in coordinated cooperative action. While the explanation in [5] focuses on a company internal context, the same question is examined here for cooperative action involving independent third parties.
Keywords: systemic approach, specification, resource, operating conditions, interfaces, components, component models, services.

1 Introduction

Besides the orientation towards contradictions, the *systemic approach* is one of the central concepts of TRIZ. In contrast to more diffuse concepts such as an "orientation towards the needs of the user" (as in Design Thinking, [14]) or a little-conditioned "brainstorming" [15], this achieves a *focussing* of the modelling in the problem solving process and later a concentration on a spatio-temporally more narrowly defined *operative zone*, such as in the ProHEAL Path Model [3] or in the classic ARIZ-85C.

The concept of a (description of a) system in TRIZ [13, p. 17], [21] is oriented towards an *emergent function* as *main useful function*, which emerges not from functionalities of a single component in the system but from their *interaction* in the systemically delimited context. The three delimitations required for this [4, p. 3],

- an external demarcation of the system against an *environment*,
- an internal demarcation of the system against *components* and
- a reduction of the relationships between the components in the system itself to *causal essential* ones,

adjust exactly the right degree of abstraction that is needed for successful problem solving.

TRIZ as a "whiteboard technology" pays little attention to the phase of *implementing* the solution itself. But "dreaming up a process on a whiteboard is one thing, making it happen quite another" as pointly remarked by Howard Smith [17]. At the theoretical-mental level the solution of an engineering or management problem remains a *conditional mind game: if* certain operational requirements are met, *then* the implementation of the proposed solution also achieves what is required in practical operation.

For this to happen, however, the thing *previously delimited* as a system at the whiteboard must be *inserted* back in the context of a "living world" at the place from which it was previously "cut out" for analytical reason. This context consists of "living" components and a "living" environment as pointed out in [16, p. 98]. Due to the self-similarity of the systemic concept both parts – the thing to be inserted and the target – can be considered as systemically structured themselves if we assume the *universality* of the principle of a systemic structure of a highly technical world that is increasingly developing based on planning.

However, this does not only raise the question of a systemic solution of a delimitable problem up to that "conditional mind game" (Shchedrovitsky calls this a *first concept of a system* [16, p. 89]). It also poses the questions not only of the *implementation*, but also of the *integration* of that solution into the *structures and order of the living world*. At this point the fundamental contradiction of every systemic solution manifests itself – the contradiction between the *necessity to decompose it* for analytical purposes and the *indecomposability* of the system in operational mode. "An aeroplane consists of many parts. None of them can fly, not even the sum of all parts. Only the aeroplane in assembled mode can fly, can be operated." This rephrases an example given in [13, ex. 1.7]. But this indecomposability in operation does not end at the boundary of the system, because the aeroplane needs fuel for its engines and an "airy" environment to fly. On the moon, a terrestrial aircraft would be of little use. In [5] using the words of Shchedrovitsky it is expressed even more pointedly: "The thing viewed with the magnifying glass as a connection of place and content remains a 'dead body', because 'a living being has no parts'" [16, p. 91].

So it is a matter of bringing the "dead body" to life, embedding it in a "living" systemic context and thus systemically *evolving* also that context itself as a "living system", because after integration a function is available there that was not available before.

In the concept of systemic composition to be developed (Shchedrovitsky calls this a *second concept of a system* [16, p. 98 ff.]), the concept of *resources* plays a central role. In the development of a system of the first kind up to that "conditional mind game", this future composition requirement must of course already be taken into account.

The (evolutional) development of a whole in a mode of production that is characterised by a deep division of labour is only possible based on cooperative action of differently qualified actors. The technical question of providing suitable resources and especially *tools* (i.e., components with a main useful function) and *workpieces* (i.e., the objects to be processed by those tools) thus becomes a socio-culturally embedded socio-technical task.

In [5] this question is analysed in more detail and the role of *component models* and *component frameworks* is elaborated for such composition in addition to components, especially in the field of Software Engineering. However, the argumentation there was limited to a division of labour approach *within* a company as a specific form of institutionalisation of cooperative action.

In this paper, it will be analysed in more detail which further dimensions open up for such composition in a system concept of second kind if this cooperative action includes collaborations of independent third parties such as developers and users, manufacturers and customers, etc.

2 "Components are for Composition"

This central formula runs through the entire book *Component Software* [20] of Szyperski. He points out that for the operational insertion of a systemic solution as "conditional mind game" into its real-world "living" context, the view of the system as a *component* is essential, even if the "supersystem" remains vague, in which the system as a component has to prove itself operationally.

The interaction between "place" and "content", which is described in more detail in [5] and has to be organised practically for the commissioning of the component, is essentially described by *specifications* of input and output interfaces. At the input interfaces, specification-compliant operating *resources* must be provided, so that the component itself really can deliver its service according to its own output specification and thus being useful as a *resource for others.*

Such a distinction between interface specification as a "place" and filling this "place" with a suitable resource as "content" is well known in the field of *human resources* with the concept of *roles.* Within the framework of a Business Process Modelling [2], roles are *defined* and afterwards to be *filled* by suitable human resources. In socio-technical systems, e.g. organisations, such role definitions have a broader structure:

- A role is a bundle of necessary *experience, knowledge and skills* that an employee must have in order to perform a certain *activity.*
- Roles are defined by *role descriptions* within a *role model.*
- A role is associated with *activities* and *responsibilities.*
- *Qualification characteristics* are required to perform a role.
- A person can have several roles. Several persons can have the same role.

It is clear that such structural concepts can easily be transferred to other types of resources. As noted above especially components and their output play a role as resources for other, causally and temporally subsequent structures and processes, these structural concepts must also be applied to components when it comes to their "composition".

Sommerville emphasises the importance of interface specification for the development of software systems that use existing components (COTS – Commercial off the Shelf) or for large systems to be developed in a cooperative process and require a decomposition into subsystems to be developed independently of each other [18, ch. 10.2].

Such component-based development scenarios are of growing importance over the last 20 years and developed to an established approach in software engineering. With the transition to such component architectures, software engineering has taken a turn to a development mode which is characteristic for other engineering domains for a long time already. Many of the phenomena compiled here from the perspective of software technology thus have a broader scope of validity and ultimately distinguish craftsmen from engineering approaches. In this way systemic development manifests itself as a concurrent process of parallel in time developments and unfolding of subsystems, which is controlled by a socio-technical supersystem of project coordination.

Sommerville [18, p. 477] emphasises that this development process in turn requires a more extensive socio-technical infrastructure with

1. *independent components* that can be fully configured via their interfaces,
2. *standards for components* that simplify their integration,
3. a *middleware*, which supports the component integration with software
4. and a *development process* that is designed for component-based software engineering.

Components are thus conceptually integrated into an overarching *component model*, which essentially ensures the technical interoperability of different components beyond concrete interface specifications and thus forms a moment of *the whole* in the diversity of the components.

This unified model is not only a *conceptual* envelope, but also a *practical* one since the middleware directly provides a certain "general" part of the operating conditions, and also the *practical development process* of such components is significantly influenced by the standardised patterns of the respective component model. The last aspect is particularly emphasised in the survey [1]

> A *component model* specifies the standards and conventions imposed on developers of components. Compliance with a component model is one of the properties that distinguish components (as we use the term) from other forms of packaged software. A *component framework* is an implementation of services that support or enforce a component model. Both are examined more closely, below. [1, p. 23]

The authors of that study on the state of Component Based Software Engineering (CBSE) initiated in 2000 by the SEI at the CMI emphasise the general root of every component-composition approach:

> The parts that we compose are, etymologically speaking, *components*. Why this pedagogy? Because, by definition, all software systems comprise components. These components result from problem *decomposition*,

> a standard problem-solving technique. In the software world, different conceptions about how systems should be organised result in different kinds of components. Thus, two systems may comprise components, but the components may have nothing more in common than the name "component". The phrase *component-based system* has about as much *inherent* meaning as "part-based whole". [1, p. 1]

But the existence of component models and component frameworks based on them points to essential inner connections and cooperative potentials of such "problem-solving techniques" and overarching concepts, which structure and simplify problem-solving and hence impose structure on the part-whole relation.

And the reverse is also true: Consolidated experiences from the development processes of components used in the real world form the basis for the further development of the component model. This frame constitutes as *component framework* [20, ch. 9] a *socio-technical supersystem* as an "environment" of components that were created according to the (abstract) specifications of that component model.

Szyperski, for his part, analyses in [20] this diversity of compatibilities and incompatibilities of *different component models* and identifies different levels of abstraction for the reuse of concepts that go beyond the use of prefabricated components. In his 20-year-old book he already emphasises

> the growing importance of component deployment, and the relationship between components and services, the distinction of deployable components (or just components) from deployed components (and, where important, the latter again from installed components). Component instances are always the result of instantiating an installed component – even if installed on the fly. Services are different from components in that they require a service provider. [20, p. xvii]

3 Components and Third Parties. The Open Source Practice

Different to [5] below it is examined how components from *different mutually independent sources* can be integrated into a system. It is less about the "conditional mind game" of writing the code than about bringing together the necessary "living" components as resources to form a "living" system.

This is particularly easy in the software sector, because "software is different from products in all other engineering disciplines. Rather than delivering a final product, delivery of software means delivering the blueprints of products. Computers can be seen as fully automatic factories that accept such blueprints and instantiate them." [20, p. 8]

In the end, it is not quite that simple, since the components as program code (even if availably already in binary form) are the different processual descriptions that must be brought together and executed on an operating unit (e.g. a computer) that itself operates a *specific* operating system. But such a situation is quite typical also for modern *material* production environments since a computer as part of the "automatic factory of control" of a real, "living" factory as supersystem – and within that supersystem extended by appropriate sensors and actuators – forms the *core* of modern "complete technical systems" in the sense of TRIZ.

In order to operate this control unit *in practice*, the software has to be "built", i.e. the "places" of its components (the "dependencies" in the IT developers' language) must be filled with appropriate "content" to obtain a running ("living") software as a whole. For this appropriate "living" components as *resources* are to be allocated. In the simplest case, such a resource is itself a piece of software, but not "anything in or around the system that is not being used to its maximum potential" [11] as D. Mann defines the concept of a resource. At this "place" a resource is required that exactly serves a qualitatively and quantitatively well-defined specification.

If such a resource is to be obtained from an independent third party, a (social) supply relationship must first be established. If we restrict ourselves to the technical dimension of such a relationship, firstly place and time of availability and provision of the component are to be negotiated. For software, this is not much of a problem today with the networked structure of the internet – one may download the resource from a suitable place.

For this purpose, even a *decentralised* system of provision of such "living" components can be built if there is a *central* place – at least virtually central as Google – where the *information* about all available software components is located and can be retrieved. In the following we describe such a system for components of a certain kind – to ensure a minimal level of compatibility –, for components written in Java. This serves as a mere example since similar distribution systems for components, i.e., technical equipment of different kind, exist in almost all technical domains and constitute large domain-specific B2B markets. The special quality in our example is only that this distribution system is not organised in a market-like way.

Such a distribution system for Java components is *Maven* [12]. It is itself a "living" technical system developed and operated by a globally distributed community within the Apache Maven Project.

> Maven's primary goal is to allow a developer to comprehend the complete state of a development effort in the shortest period of time. [...] While using Maven doesn't eliminate the need to know about the underlying mechanisms, Maven does shield developers from many details. [...] Maven builds a project using its project object model (POM) and a set of plugins. Once you familiarise yourself with one Maven project, you know how all Maven projects build. This saves time when navigating many projects. [12]

Maven supports the build process of a software, i.e. the production process that produces a living application from a (component-based) software provided as a "conditional mind game". This production process itself is initially a "conditional mind game" since the instructions are first to be written down as *plan*,

- where to *find* the right components of the appropriate version in the globally distributed structure ("resolving dependencies"),
- to *upload* them to the own local private Maven repository (if they are not already there),
- do the same for the dependencies of the dependencies, etc.,
- when everything is successfully collected (*deployed* in the IT developers' language), start the actual *build* process with the Java compiler (i.e., the "production process" in its narrower meaning),
- if possible or required, run various specially specified *tests* on the "finished product" (i.e., provide for quality assurance),
- *install* it in its operating environment
- and then *configure* it for operation.

Afterwards this plan has to be *processed* with all possible pitfalls that might occur processing a plan.

All these steps are supported by Maven, a "command line tool written in Java" [12] for building Java applications. The *Maven engine* downloaded to a computer as a "conditional mind game" itself is an *instance of a component* of that (global and publicly available) project management system that can be "brought to life" by a Java runtime environment. It is at the same time a "useful product", a machine (the very instance that executes the *"mvn build"* command) producing a machine (the Java bytecode of the application), which itself controls a machine as part of the control component in a real-world technical system.

As pointed out in [5], in these multiple changes from "dead" *places* to "living" *contents*, the "plug" must not only fit into the "socket", but the socket must really provide the promised performance. This is itself rich of prerequisites and requires not only the reproduction and further development of a *functional* infrastructure, but also the *management of the corresponding resources*. The success of Java is rooted not so much in the *functional* properties of the language, compile and build tools, but primarily in the availability of components in software libraries for a wide variety of routine tasks as *pool of resources* of complicated structure. In this sense, Maven is a technical system for managing and operating this resource pool by a socio-technical supersystem that would not and could not function without the cooperative actions of interested and motivated developers on its part.

Maven is not the only example and not even the flagship of such cooperatively developed and operated component frameworks, but a typical form of cooperative governance in Open Source practices, as reveals the long list of other Apache Projects at https://www.apache.org.

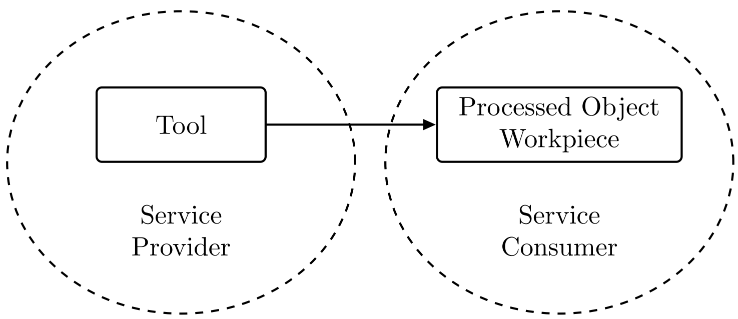
4 Services

In the previous section, we analysed cooperative practices in CBSE in which the materialisation of the component as "living" part of a whole (the finally running software) ultimately realises as code on a concrete computer. This code then has to be "put into operation", i.e. called up by another tool, the "operating system" of that computer, in order to unfold the desired processual effect.

In this context the function remains bound to the tool. The independent third party must provide the tool or at least – as in the Open Source practice described above – support the user in producing the tool in order to apply its main useful function to his objects (workpieces).

In a mode of production based on the division of labour, less is conceivable. Why not ask (and pay) the third party for applying that main useful function to the user's objects (workpieces) with their own tools? In this way, too, the state-changing effect is exerted on the objects in the user's possession. At this point, however, the function is not needed as a *pure function* with *potential* usefulness, but as a "living" function with a real world-changing effect, as a *service* of a *service provider*.

We are thus getting closer to the *principle of the Ideal Machine* as "a solution in which the maximum utility is achieved but the machine itself does not exist" [7, p. 40]. Within such a concept of services the machine remains at least invisible to the service consumer like the origin of the flying roasted chickens in that land of milk and honey. The problems, however, shift to the next cooperative level, to the socio-technical supersystem in which this business relationship takes place.



In fact, the essential problem of such an approach is not so much a technical as a social one. Even the minimal in the TRIZ understanding technical system of this cooperative action extends over *different* areas of responsibility as displayed in the figure above. It requires clarification how problems and errors in the operation of such a cooperative structure are to be dealt with, who has to compensate the damage claims, but also how the "value proposition" [19] resulting from the cooperative action is to be distributed.

Such a service-oriented approach [8], which software development discovered for itself about 20 years ago with the slogan "Software as a Service", is, however, much older when looking at the achievements of a networked technical world: water comes from the tap, electricity from the socket, groceries from the mall and Twitter messages from the smartphone. It is claimed that modern people would have great difficulty to survive for more than two weeks if all these things we are taking for granted were to disappear at once.

Service-oriented architectures live from the fact that the infrastructural reproduction processes of resource provision and management are organised in an appropriate cooperative manner. Only on this viable structural basis component-based TRIZ (and non-TRIZ) solutions are possible at all. Garrett Hardin [6] argued that a cooperative resource management as infrastructure is alien to the benefit maximising *homo oeconomicus*. The conditions for successful *socio-cultural management of resource pools* do not arise on their own but must themselves become the target of a consciously designed cooperative systemic development process.

References

1. Bachmann, F., Bass, L., Buhman, C. et al.: Volume II: Technical Concepts of Component-Based Software Engineering. Technical Report CMU/SEI-2000-TR-008, ESC-TR-2000-007.
2. Caetano, A., Ritó Silva, A., Tribolet, J.: Using roles and business objects to model and understand business processes. Proceedings of the 2005 ACM symposium on Applied computing (2005), pp. 1308–1313. DOI: 10.1145/1066677.1066973
3. Gräbe, H.-G., Thiel, R.: ProHEAL – Social Needs and Sustainability Aspects in the Methodology of the GDR Inventor Schools. LIFIS Online, 15.08.2021. DOI: 10.14625/graebe_20210815
4. Gräbe, H.-G.: Technical Systems and Their Purposes. In: Mayer, O. (ed.): Proceedings TRIZ-Anwendertag 2020. pp. 1-13. Springer Nature (2021). DOI: 10.1007/978-3-662-63073-0_1
5. Gräbe, H.-G.: Systems and Systemic Development in TRIZ. Submitted to TFC 2022.
6. Hardin, G.: The Tragedy of the Commons. Science 162 (1968), pp. 1243–1248. DOI: 10.1126/science.162.3859.1243
7. Koltze, K., Souchkov, V.: Systematic Innovation Methods (in German). Hanser (2017).
8. Laskey, K.B., Laskey, K.: Service oriented architecture. WIREs Comp. Stat. (2009), pp. 101–105. DOI: 10.1002/wics.8
9. Li, J. et al. (2005). An Empirical Study on Off-the-Shelf Component Usage in Industrial Projects. In: Bomarius, F., Komi-Sirviö, S. (eds). Product Focused Software Process Improvement. PROFES 2005. Lecture Notes in Computer Science, vol 3547. Springer, Berlin, Heidelberg. DOI: 10.1007/11497455_7
10. Lyubomirskiy, A., Litvin, S., Ikovenko, S., Thurnes, C.M., Adunka, R.: Trends of Engineering System Evolution (TESE). TRIZ Consulting Group (2018).
11. Mann, D.: Hands On Systematic Innovation. IFR Press 2014.
12. Apache Maven Project. https://maven.apache.org/what-is-maven.html
13. Petrov, V.: Laws and patterns of systems development (in Russian). Independent Publishing (2020).
14. Razzouk, R., Shute, V.: What Is Design Thinking and Why Is It Important? Review of Educational Research (2012), Vol. 82, No. 3, pp. 330–348. DOI: 10.3102/0034654312457429
15. Ritter, S.M., Mostert, N.M.: How to facilitate a brainstorming session: The effect of idea generation techniques and of group brainstorm after individual brainstorm. Creative Industries Journal, 11:3 (2018), pp. 263-277. DOI: 10.1080/17510694.2018.1523662

16. Shchedrovitsky, G.P.: Selected Works. A Guide to the Methodology of Organisation, Leadership and Management. In: Khristenko, V.B., Reus, A.G., Zinchenko, A.P. et al.: Methodological School of Management. Bloomsbury Publishing (2014).
17. Smith, H.: P-TRIZ in the History of Business Process. A BPTrends Column, April 2006.
18. Sommerville, I.: Software Engineering. Citations based on the 8th German edition. Pearson Studium (2007).
19. Souchkov, V.: TRIZ and Systematic Business Model Innovation. In: Proceedings TRIZ Future Conference 2010, Bergamo, Italy.
20. Szyperski, C.: Component Software. 2nd edition. ACM Press (2002).
21. The TRIZ Ontology Project. https://wumm-project.github.io/Ontology.html.