Patterns in Reactive Programs

P@trik Haslum[1]

Abstract. In this paper, I explore the idea that there are "patterns", analogous to software design patterns, in the kind of task procedures that frequently form the reactive component of architectures for intelligent autonomous systems. The investigation is carried out mainly within the context of the WITAS UAV project.

1 Introduction

A current trend in AI research is the focus on *autonomous systems*: robotic, or "softbotic", systems capable of acting independently and intelligently, in uncertain, varied and dynamic environments. The wish to make agents act intelligently and the requirements imposed by the dynamic nature of the environment combine to create competing needs for *deliberation* and *reaction*, and much attention has been given to the design of system architectures that somehow mediate between these needs. The solutions that have, to date, appeared most successful are variants of so called "layered architectures", where deliberative, reactive and low-level process components run and exchange information asynchronously, *e.g.* [10, 1, 13].

Part of every layered architecture variant is a reactive system, consisting of a base of *procedural knowledge*, rules that prescribe reactions to standard situations or flexible "scripts" for enacting standard tasks, and a mechanism that translates the encoded knowledge into action (*i.e.* commands to low-level system processes), contingent on the state of the environment as perceived via the systems sensors. The core of any reactive system is the procedural knowledge base, and there have been many proposals for languages to express such knowledge. Less attention has been devoted to the problem of how to fill the procedural knowledge base with *content*. The task of writing the actual rules or programs that the reactive layer will execute is often an ardous one, and there are few principles or guidelines to help the programmer who must in the end perform it[2] (notable exceptions are discussed in section 6).

By contrast, principles of programming and design have since long been studied and developed in other areas of computer science, often without reference to the particulars of any implementation language. It may be worthwhile to look likewise at the problem of writing reactive programs. This paper presents a tentative step in this direction: By examining examples of reactive programs, from the literature and from experiences in the WITAS autonomous UAV project, I try to find something akin to "design patterns" [7] for reactive layer procedures. Whether the things found actually qualify as design patterns (or, indeed, have any value or meaning at all) is debatable. This little effort should perhaps be viewed more as a sign that *something* is missing, rather than a definitive answer to what that something should be.

The next three sections provide brief background on languages for programming reactive procedures (section 2), the WITAS project and system architecture currently employed within it (section 3) and design patterns (section 4). Section 5 describes and illustrate some reactive design patterns, while section 6 contains some concluding discussion.

[1] Linköpings Universitet (pahas@ida.liu.se)

[2] Some systems, *e.g.* CIRCA [14] or the situated automata of Kaelbling and Rosenschein [9], synthesize the reactive procedures from a specification of the task to be achieved and a model of the environment. This, for better or worse, moves the problem from programming to specification and modelling, but at the same time leads to some restrictions on the expressivity of the task specifications, environment model, and the procedures that the system is able to generate, to make the synthesis problem decidable.

2 Rule and Reactive Procedure Languages

A large class of reactive programming languages consist of *condition – action* (or *event – condition – action*) rules. Such rules prescribe an action to be taken whenever the corresponding condition on the systems view of the environment holds. As the complexity of tasks and environment grows, two problems with rule-based programs arise: In a given situation, the condition part of several rules, advising conflicting actions, may hold, or no rules condition may be fulfilled, leaving the system without a response.

A common solution to the first problem is to introduce an *arbitration* mechanism that decides which rule takes precedence in each situation. An example of this approach is the context-dependent blending method developed by Saffiotti *et al.* [15]. In this system, the antecedent of a rule is a combination of (fuzzy) predicates, while the consequent specifies for each possible action how desirable that action is, from the point of view of the rule. Collections of rules are grouped into behaviors, associated with a (fuzzy) context conditions. The desirability assigned to each action by a rule is modified by the degree to which the conditions of the rule and the behavior it belongs to are satisfied, and the action that receives the most support, overall, is the one taken. A different solution, examplified by Lin [11], is to perform a static analysis on the rule base and verify that a conflict can not occur.

Reactive procedures were introduced into AI circles mainly with the PRS system [8]. The RAPS [5] system is similar, and serves as an illustrative example of the approach. A task procedure in RAPS has a "signature", consisting of name and arguments, a context condition, a success condition and a procedure body which is a partially ordered set of steps that may be calls to subtasks, primitive actions, or "wait for condition" statements. RAPS allows having several different procedures for the same task and this is what lends flexibility to their execution: When a call for a particular procedure signature is made, the context conditions of all matching procedures are evaluated against the systems current knowledge state, and among those whose conditions are satisfied, one is chosen for execution. Eventually, the procedure either finishes successfully (its success condition becomes satisfied) or fails. If the chosen procedure failed, the system may re-evaluate context conditions to choose another procedure (or even to try the same procedure again) or the call as a whole may fail.

3 The WITAS Project and Architecture

The WITAS project aims to develop architechtures and technologies for intelligent autonomous systems in general, and for an autonomous Unmanned Airial Vehicle (UAV) for traffic surveillance in particular[3]. This choice of platform and application area leads to many challenges, but also to a problem that, overall, is in the realm of the possible.

The current WITAS system architecture is the result of many development iterations. Two important characteristics that have emerged are that it is distributed, and that the reactive system plays a central and "driving" role. A distributed architecture is advantageous for several reasons:

- Different system components have different needs: The UAV controller operates under hard real-time constraints, parts of the image processing system may need to run on specialized hardware to achieve acceptable performance, *etc.* In addition, the limited power and payload capacity of the UAV may force some components to reside in a ground-side part of the system, while still interoperating smothly with those components that reside on-board the UAV.
- Distribution lends a certain fault-tolerance, since separate system components can be restarted (or even rebooted) in case of failure, without the need to bring the whole system down.
- An interesting area for future research is the integration of several UAVs, and possibly also other actors such as ground stations (manned or unmanned), into a system capable of acting coherently and efficiently. A distributed architecture leaves many more "hooks" for development in this direction.

To minimize the extra complexity introduced by distribution, a choice has been made to use a CORBA infrastructure[4]. The use of CORBA carries some additional advantages, such as for instance simplifying the transition from a simulated to a real environment.

The "reactive-centric" nature of the architecture is in part an effect of the fact that is distributed, and of the use of CORBA: At the level of interfaces, there is simply not that much difference between *e.g.* the UAV flight control system and a high-level deliberative function such as prediction or a GIS database, and thus the different components are naturally viewed as a collection of "services" for the reactive systems use.

3.1 The Modular Task Architecture

The reactive system, like the rest of the WITAS architecture, has been through a number of iterations. The current version, tentatively named the Modular Task Architecture (MTA), is, also like the rest of the WITAS system, distributed, and for pretty much the same reasons.

[3] For a more detailed description of the project, see Doherty *et al.* [4] or `http://www.ida.liu.se/ext/witas/`

[4] The Common Object Request Broker Architecture (CORBA) is an object-oriented middelware standard laid down by the OMG (`http://www.omg.org/gettingstarted/corbafaq.htm`). The interfaces of CORBA objects are specified in the Interface Definition Language (IDL), which maps to whatever language is used to implement the object.

The common denominator shared by all MTA task procedures is a CORBA interface and a few behavioral restrictions. The task interface is rather basic, containing only operations such as passing arguments, starting and canceling a task. Asynchronous messages are sent from a task to its caller via event channels[5], and a few message types, *e.g.* those signalling task completion or failure, are standardized. This simplicity, however, should not be mistaken for a limitation. Beyond the requirements that MTA places on a task, each task procedure is free to react to events in any form and interact with any component of the WITAS system that is accessible through an interface.

Because the MTA is, in essence, only a standard, task procedures can be implemented in any language (for which there exists CORBA support). Indeed, they have to. As it has turned out that large parts of most task implementations tend to be routine excercises in CORBA programming, we have developed a simple macro language and translator to make writing tasks easier and less prone to cut-and-paste errors[6].

4 Design Patterns

There are numerous definitions of what constitutes a design pattern (or just "pattern"). The term was originally used by architect Cristopher Alexander, who has written several books to explain what is meant by it. Gamma, Helm, Johnson & Vlissides, whose book on object-oriented design patterns has probably done most to popularize its use in software engineering, write: "A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. [...] Each design pattern focuses on a particular object-oriented design problem or issue. It describes when it applies, whether or not in can be applied in view of other design constraints, and the consequences and trade-offs of its use." [7]. A shorter, less object-oriented, description is "A pattern is a named nugget of insight that conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns" [2]. Patterns have been recognized at an "application" (or "architectural") level, at the design level, and at the "language" level, *i.e.* in program constructs (where they are often called "idioms") [2].

5 Patterns in Reactive Programs

This section describes five different patterns that I have seen in reactive task procedures. The first two examples presented here are more "architectural" in flavour, describing what is essentially structures in the procedural knowledge found in an application domain, while the remaining examples are more design oriented. The first two are also found in the structure of a single task procedure, while the other concern the interplay between two or more tasks.

5.1 Scripts

In the introduction, task procedures were described as "flexible scripts". This is a frequently occuring form, the task procedure consisting of a set of steps to be carried out, in sequence or just in partial order, interspersed with waiting for events or conditions.

[5] Channels are specified in the CORBA Event Service standard (`http://www.omg.org/technology/documents/corbaservices_spec_catalog.htm`), and allow decoupled passing of arbitrary data from one or more senders to one or more receivers.

[6] The language, called TSL, is based on XML. The translator consists basically of an XSLT processor and a library of templates for each implementation language used.

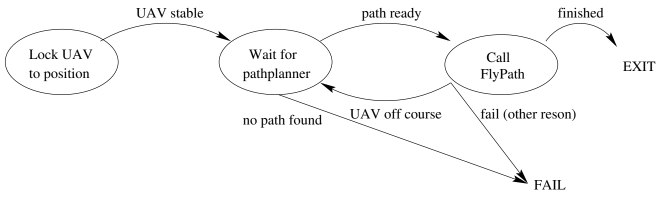

Figure 1. The `NavToPoint` Task Procedure

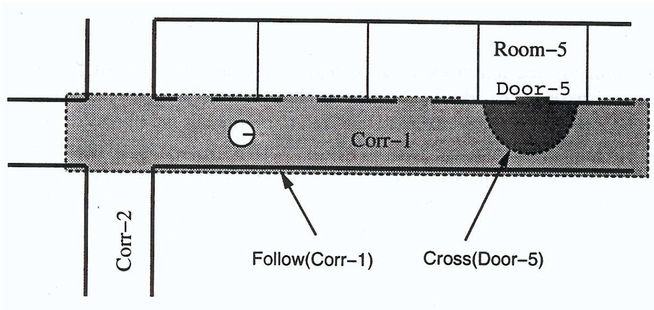

Figure 2. The `follow` and `cross` behaviors combined into a script. Reprinted from Saffiotti *et al.* (1995).

Figure 1 shows a schematic of a task procedure for safely navigating the WITAS UAV to a goal position. The normal execution of this task forms sequence of three steps, corresponding to the three states: First, the UAV is stabilized (locked onto its current position), then a request for a safe trajectory from this position to the goal is made to the pathplanning service, and finally a subtask, `FlyPath`, is invoked to execute the trajectory returned by the pathplanner. The possible exception to this normal case is if any of the steps fails, *e.g.* if the pathplanner can not find a flyable trajectory, or if the `FlyPath` task fails. In most cases, this leads to the task as a whole failing, but in one certain case, when `FlyPath` fails because the UAV has drifted too far off course, it only causes the task to "back up" and try again (the reason why it does not back up all the way to the first state is that the `FlyPath` task locks the UAV into a stable hovering mode when failing in this way).

Script-like task procedures can appear also in rule-based systems, though they may be less obvious. The following example of a script implemented by fuzzy behaviors (due to Saffiotti *et al.* [15]), involves a mobile robot navigating in an office environment: The robots goal is to enter a certain room, which it may achieve by a behavior `cross(doorway)`. The context of applicability of this behavior, however, is limited to the close vicinity of the doorway. Another behavior, `follow(corridor)` is applicable in any part of the corridor, and will when effected lead the robot down the corridor, eventually to reach the doorway. A procedure to achieve the goal from the wider context is created by conjoining to the context of `follow` the negation of the context condition for `cross`, and applying the context-dependent blending method to the two behaviors. The result acts like a script, applying first the `follow` behavior until a situation within the context of `cross` is reached, then the `cross` behavior. This is illustrated in figure 2.

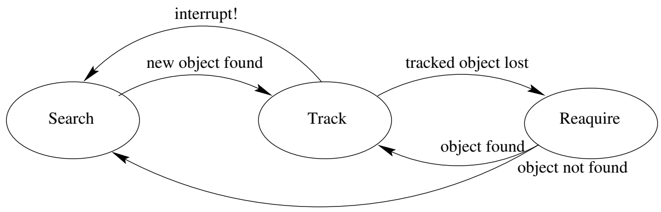

Figure 3. The `FindTrack` Task Procedure

5.2 Mode Switchers

While many task procedures take the form of scripts (though in some cases more elaborate, *e.g.* with alternate branches for different conditions), some are very definitely not of this kind. Another common cathegory are "mode switchers": tasks that continuously change between a set of different working modes, depending on circumstances, and that often do not have any wired-in terminating condition but carry on indefinitely, until interrupted from without.

Figure 3 shows an example of such a task procedure for the WITAS UAV, which uses the image processing system to alternatly search for objects (defined either by motion or by color) in the camera image, and tracking an object for as long as possible. It has three modes: Searching for an object, tracking the object once found, and "reaquire", which is entered immediately upon losing track and in which the task searches for an object similar to the one just lost near the objects last known image position, for a (short) limited time. While in the tracking mode, an interrupt command causes the task to drop the currently tracked object and resume search (a different external command causes the task to terminate, but this is not illustrated in the figure).

5.3 Fail and Retry

Reactive procedures are typically designed to accomplish a particular task, in a limited range of operational circumstances. This limitation is key to keeping the complexity of individual procedures managable. Were we to try to deal with every imaginable contingency that may arise in carrying out a desired task, procedures would quickly become unwieldingly complex. Also, the appropriate response to an abnormal situation may vary depending on the overall plan that the task is part of. For this reason, most reactive systems have a notion of task procedures *failing*. In *e.g.* RAPS or the WITAS system, failures are signalled explicitly, while in Saffiotti *et al's* system a behavior may be defined as "failing" when the degree of satisfaction of its context condition becomes too low.

A pattern related to failing is "catch and retry". It involves two task procedures, one of which ("the caller") has invoked the other ("the callee") and is waiting for it complete. There are many reasons that may cause the callee to fail, but for some of them, the caller can take measures to repair the failure, appropriate to the purpose that the caller called the callee for. Thus, if the callee fails, the caller determines the reason for the failure, and if it is of a kind that the caller knows how to deal with, it takes some action to remedy the problem and restarts, or calls again, the callee. If the failure is of any other kind, the caller itself fails.

An example of this pattern has already been shown, in the `NavToPoint` task procedure: If the `FlyPath` task fails due to the

UAV drifting off course, the procedure solves the problem (by asking the pathplanner for a new path, from the current position and invoking `FlyPath` again). Note that, again, this may not be the right way to deal with the problem in all contexts. If, for example, a preprogrammed path was flown to record sensor data, the flight may have to be started over from the beginning.

5.4 Supervision

Task failures is only one half of a two-sided problem. In some situations, a task procedure may be acting inappropriately without realising it, thus *not* failing when in fact it should.

A way of dealing with this situation is *supervision*. Again, this involves two task procedures, a "caller" and a "callee". The callee, during execution, sends "status reports" to its caller, who through monitoring these and the state of the environment may detect when the callee is responding incorrectly. The caller may then interrupt the callee, or send to it some corrective commands.

The `FindTrack` task described above frequently acts as callee in this kind of pattern. When it is invoked, it is for a purpose, *e.g.* to find and track a particular vehicle, but the `FindTrack` task can not discriminate the vehicle of interest from others that may be found, or even discriminate vehicles from other moving things. Therefore, the task reports to its caller whenever it changes from searching to tracking the identity of the tracked object. The calling task may then retrieve information about this object and apply more sophisticated reasoning to determine if it is one that should actually be tracked (*e.g.* matching the movement of the object with information about the road network in the area to determine if it follows the road or not). If it is not, the `FindTrack` task can be commanded to drop the object and switch back to searching.

The same result could, of course, be obtained by implementing the `FindTrack` functionality in its caller, adapted to the task that the caller performs, but separating the two confers several advantages: It avoids code duplication (*i.e.* writing and debugging the same program twice), since the `FindTrack` task procedure can be used by many different tasks. It keeps the calling task procedure simpler, since it does not have to handle the idiosyncracies of interfacing to the image processing module and since the tracking of an object operates concurrently with the (possibly time-consuming) reasoning performed by the calling task, without the need to write this concurrent handling into the calling task explicitly.

5.5 Higher-Order Task Procedures

In more complex task procedures, there is often a heirarchical structure: the task decomposes into a series (or set) of subtasks, with some coordinating or "bridging" activity between them. Sometimes, for a group of tasks this bridging part may be the similar, or even identical, even though the tasks in the group are applicationwise unrelated. For example, two potential tasks for the WITAS UAV are surveying a collection of buildings (or other structures of interest) scattered throughout an area, and searching an area for a particular vehicle. Both these tasks involve navigating the UAV to a series of positions in turn (positions of the buildings in the first case, positions where the sought vehicle is likely to appear in the second), and performing some data-gathering activity at each position (taking photographs from different angles in the first case, image/video analysis in the second).

A single task procedure could be written to handle both tasks (as well as other tasks with similar structure), by using enough parameters to define the data-gathering activity that has to be done at each position. However, this procedure will grow very complex and difficult to maintain as the set of possible data-gathering subtasks grows. But, since the navigation part of the overall task is (mostly) independent of the activity carried out at each position, an alternative is to write a "higher-order" task procedure, `DoAtPositions`, which takes as argument a set of positions and an arbitrary task to carry out at each position[7]. Again, this both simplifies the writing of the task procedures involved and improves the potential for reuse.

6 Conclusions

These ideas, although grounded in some experience, are speculative. Here are some objections that can reasonably be made:

6.1 "This is all very interesting, but hardly new."

Although the concept of design patterns in software was introduced not so many years ago [7], there has been an almost explosive development in "pattern recognition" since[8], and there are even collections of patterns specifically aimed at the kind of concurrent, distributed programming that is typical of MTA task implementations [16, 12]. Shouldn't the simple observations found in the preceeding section already have been made, many times over? Indeed, they have. But this lack of novelty is not a fault, since one of the hallmarks of a good pattern is recurrence, in varying contexts.

Also, a few AI researchers have discussed reactive programming practice: Firby [6] describes a collection of RAPS task procedures written for an in-door mobile service robot, and atempts to structure it into modular, reusable subtasks. His conclusion is that heirarchical task decomposition, while a powerful structuring principle, alone is not enough. Some tasks need to spawn subtasks whose execution is tied to a condition on the state of the robot or its environment, and thus streches beyond that of the spawning task (this is perhaps also a candidate for a pattern). Beetz [3] analyzes reactive plans (programs) for mobile robot navigation tasks and designs a representation language specific to this application by introducing constructs that match patterns of use. Examples of at least two of the patterns discussed in the preceeding section can be found among those: supervision (expressed as augmenting default plans with context-triggered subplans) and higher-order tasks (expressed by the "at location" macro, which specifies that a particular part of the plan needs to be executed at a certain position, abstracting the details of how to get the robot to the position from the task to be carried out there). Beetz also introduces the notion of a task procedure being "embeddable" (meaning it can be safely run concurrently with other tasks, even in the presence of conflicting resource needs), "interruptible" (meaning it can be interrupted/resumed at arbitrary points without compromising the procedures ability to complete its task) and "transparent" (meaning the procedure accomplishes one and only one goal, and that this goal is explicitly indicated), and argues that these are all desirable properites to form a library of reusable task procedures.

6.2 "This is all very interesting, but what's the point?"

There are several:

[7] The mechanism used for passing subtasks as arguments depends on the language used to implement the task procedure. In the MTA framework, a task can be passed as a CORBA object, or a specification of the task (name and arguments) can be passed as a data structure definable in IDL.

[8] As evidenced by pattern catalogs, *e.g.* at `http://hillside.net/patterns/`.

Patterns suggest good practices: A task procedure that provides information on the reason for failures is more reusable than one that does not, since it can be combined with other tasks in a catch-and-retry fashion. Likewise, a mode-switching task procedure that provides meaningful status reports and "hooks" to force mode changes can be supervised, and therefore more useful as a subtask. A common theme in all the three design-oriented patterns in the preceeding section is that they aim towards increasing the potential for *reuse*, which is a cornerstone of efficient (or economic) software construction.

Patterns of use motivate features of task procedure languages in existence, and suggest potentially useful features missing from languages of today. Beetz design of a representation for robot navigation tasks [3] is an example of this approach. In creating the reactive layer of a layered architecture, looking for patterns in the intended application domain(s) may also guide the choice of implementation technology. In the WITAS UAV project, we have found a mode-switching structure to be more common than a script-like one, at least among basic tasks having to do with control of the UAV platform and its sensors. In an early phase of the project, the reactive layer was implemented using the RAPS language, and one of the lessons learned from this was that although it is certainly *possible* to implement a mode switching task using the constructs of this language, it is not very convenient.

6.3 "This is all very interesting, but it should be formalized."

Probably the most important function served by patterns is as an educational resource: They communicate experience, insight, and sometimes inspiration, between people faced with similar problems (programmers, in the case of software patterns). Thus, it is more important for a pattern description to be human-readable than machine-readable. However, recurrent patterns in reactive programs possibly also point towards mechanisms for automatically synthesizing such programs, for example in the form of search spaces or search control knowledge for automated planners (this is also advocated by Beetz [3]).

Acknowledgements

Thanks to the reviewers for pointing out Firby's work, and for raising some of the objections. The WITAS project is, of course, a team effort, but I would especially like to mention Per Nyblom, who is the author of the TSL translator, and who has also contributed much to the development of the library of task procedures for the WITAS UAV. This work is partially supported by research grants from the Wallenberg Foundation, Sweden and NFFP-539 COMPAS.

References

[1] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand, 'An architecture for autonomy', *International Journal of Robotics Research*, **Special Issue on "Integrated Architectures for Robot Control and Programming"**, (1998).

[2] B. Appleton. Patterns and software: Essential concepts and terminology. `http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html`, 2000.

[3] M. Beetz, 'Plan representation for robotic agents', in *Proc. 6th International Conference on Artificial Intelligence Planning and Scheduling (AIPS'02)*, (2002).

[4] P. Doherty, G. Granlund, K. Kuchcinski, E. Sandewall, K. Nordberg, E. Skarman, and J. Wiklund, 'The WITAS unmanned aerial vehicle project', in *Proc. European Conference on Artificial Intelligence*, (2000).

[5] R. J. Firby, *Adaptive Execution in Dynamic Domains*, Ph.D. dissertation, Yale University, 1989.

[6] R. J. Firby, 'Modularity issues in reactive planning', in *Proc. International Conference on AI Planning Systems (AIPS'96)*, (1996).

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[8] M. P. Georgeff and A. L. Lansky, 'Procedural knowledge', in *Proc. IEEE Special Issue on Knowledge Representation*, volume 74, pp. 1383 – 1398, (1986).

[9] L. P. Kaelbling and S. J. Rosenschein, 'Action and planning in embedded agents', in *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*, ed., P. Maes, MIT Press, (1990).

[10] K. Konolige, K. Myers, E. Ruspini, and A. Saffiotti, 'The Saphira architecture: A design for autonomy', *Journal of Experimental and Theoretical AI*, (1996).

[11] M. Lin, *Analysis and Synthesis of Reactive Systems: A Generic Layered Architecture Perspective*, Ph.D. dissertation, Linköpings universitet, 1999.

[12] T. Mowbray and R. Malveau, *CORBA Design Patterns*, Wiley, 1997.

[13] N. Muscettola, P. Nayak, B. Pell, and B. C. Williams, 'Remote agent: To boldly go where no AI system has gone before', *Artificial Intelligence*, **103**, (1998).

[14] D.J. Musliner, E.H. Durfee, and K.G. Shin, 'World modeling for the dynamic construction of real-time control plans', *Artificial Intelligence*, **74**(1), 83 – 127, (1995).

[15] A. Saffiotti, K. Konolige, and E. H. Ruspini, 'A multivalued logic approach to integrating planning and control', *Artificial Intelligence*, **76**, 481 – 526, (1995).

[16] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, Wiley, 2000.