Refine rather than design!

Andrej Lúčny

Institute of Informatics, Comenius University, Bratislava, and MicroStep-MIS, SLOVAKIA

Abstract. We introduce an approach to system development applicable in artificial intelligence domain. It is strongly based on the subsumption architecture invented by R. Brooks in half of eighties. It could be said that the approach is an application of the technology originated in mobile robotics to software engineering. However, a new explanation based on the multi-agent systems consisting of so-called purely reactive agents communicating indirectly through their environment is presented. Moreover some new issues related to a domain of software are stated. Finally some philosophical aspects are discussed.

1 Introduction

There are some crucial features what today's software technology cannot provide in spite of feeling that these features are typical for living systems like human beings are. Even we have not understood living system enough to specify the difference exactly. May be, the difference is both sharp and vague accordingly to the level of description what we have used. Anyway when it is the vague one, i.e. we do not need a new component but a better arrangement of the available ones to overcome it, we can use an original approach invented by R. Brooks in half of eighties, so-called "subsumption architecture" [3]. We do not give description of the original approach here, because it is many years old, it is specialized to domain of mobile robotics and we deal with completely translated explanation today. However the idea is the same: we can build a system before we have found its detailed specification and structure and reveal the details by building the system. This is in strong contradiction to today's standard procedure of building software where it is preferred to elaborate detailed structure of the system before its implementation. We do not dispute the manner but we take for granted that the technique cannot be applied while he have not detailed specification. Since we have already outlined that we have not clear image of the difference between living and (today's) artificial systems, it seems to be inevitable to try different approach. However, it should not be only different, but should be also reasonable. It appears to be reasonable only under certain non-trivial conditions. Therefore, we specify the conditions precisely in this article and describe an software architecture holding the conditions what we have chosen and tested.

2 Bottom-up incremental development

Each software engineer knows that the only correct method of software development is the top-down method and there are very serious arguments supporting this opinion. However, if we build very complex system, we have to do it incrementally, i.e. we have to alternate coding with operation tests of the already implemented part. Principally, the implemented system consists of many parts and each part is fundamental one or it utilizes some other parts. While we use the top-down development, we implement a part before the parts it utilizes. Therefore, during the test of operation we have to replace some utilized but not implemented parts by similar, but trivial replacements. It works well when we have exact knowledge about the behaviour of the replaced parts and we can relay that we will be able to implement the parts as planned. Further friendly condition for the top-down development is that the behaviour of each replaced part is not too complex, i.e. it is easy to implement the trivial replacement and it provides a good approximation. Any violation of the two conditions decreases the testing quality. As a result the global correctness is not guaranteed and implementation does not pursue the specified goal, hence the testing is partially or completely worthless.

We doubt that the two conditions are hold within the research projects in artificial intelligence domain dealing with complex systems like robots acting in natural environment or their simulators. Even we could say that they would not be hold also within usual software projects if production of behaviour as reasonable as the human one is required. But it would be silly to say now: "Let's use the bottom-up development instead!" Although the bottom-up method supports incremental development well, since all utilized parts must be implemented before the utilizing ones, the development method brings disadvantages much more serious than the problems mentioned with the top-down incremental development. It is well-known that there is a serious danger that we can develop a part we are not able to incorporate to the whole system later. Such part offers an interface to call, but it does not fit our needs on the higher level, though there is something related to our wishes inside the part.

There are two tricks to prevent this danger in the subsumption architecture. The first one prevents inappropriate deployment of code among implemented parts and reside in atypical decomposition of system to parts, so-called decomposition by activity. The second one prevents implementation of inappropriate interface of the implemented parts and it is the crucial idea of the presented approach. Instead usual calls, where one code part utilizes the other by putting arguments to a prepared interface and getting returned values, the approach offers monitoring and affection of the utilized part. Thus the utilizing part acts as an intruder instead of an user.

3 Decomposition by activity

This kind of decomposition prefers to design structure of the behavior produced by the system instead of structure of its code. Designer has to define individual

activities of the system what he intends to build and state order of their implementation. Each activity should be simple enough to be implemented at once, within one step of the incremental development. This approach is advantageous because designer does not need to know during analysis, how to implement the code, which realizes the activities. He simply modifies and tests a code related to a new activity until a sufficient solution is revealed within implementation phase. He starts from a scratch and modify it many times to prevent negative results of testing.

In this way we can exploit advantages of the bottom-up development to achieve a goal what we failed to achieve by usual approaches. Even we can conceive this approach as a method for implementing prototype which can be expressed later in usual code form. In this way we can overcome lack of our capability to express all details of the problem formally; it is enough to be able to recognize whether the system prototype behaves as required or not. For example if we are building an "intelligent" system, generally we do not know, what does the "intelligent" mean, but we are able to recognize whether the system produces an non-intelligent behaviour during its testing or not.

4 Intruding instead calls

Because of the above-mentioned problem of the bottom-up development, the utilization of the formerly implemented activities by the currently implemented one is enabled by the intruding. Of course the arrangement of code realizing individual parts must be chosen to support it. Regardless to the non-traditional code arrangement, the only important fact is that there is no specialized interface to enable cooperation of the higher-level and lower-level activities. Therefore the way how to utilize the lower-level part (i.e. formerly implemented one) is chosen during implementation of the higher-level part. Thus the problems typical for the bottom-up development are overcome.

5 Indirect communication

As we have stated above, the intruding requires non-traditional code arrangement. Originally in subsumption architecture it was realized by putting special devices, suppressors and inhibitors, on wires connecting finite state automata modules. However, it is important only that the code is divided to communicating modules and a module from higher level can monitor, erase or replace messages traveling between lower-level modules. Convenient concept for such relations is indirect communication where each two entities have to communicate by messages through another entity, called environment. The messages are not labeled by address of its receiver, but only by name of the place in the environment where they are stored. Thus the environment serves as black board for the code entities where they can write, read or erase their messages to agreed position. Each such position is named and the name is only reference used by code entities to address it (let us call such named position a block).

Under platforms with message passing and multitasking we can think of the discussed code modules and the environment as communicating processes. In this model the environment is a specialized server for interchange messages and the code modules are its clients. Moreover all client-server interfaces are normalized and the norm is extremely simple and tight.

Within this concept we can provide the intruding in the following way. Let there are two modules in the lower level of our system. The first one writes a message to a block and the second one reads it. Each higher-level module can read the block, erase or rewrite it. In this way it can be informed about the processes in the lower level. Using these information it can inhibit processes in the lower level or exploit the lower-level modules to coordinate their behaviour with the higher level as well.

Although using this concept of communication can provide the intruding, the data flow from one code entity to another one is more complicated. The main problem is how the receiver of message can know when to read the block providing information written by the sender. In our approach we have solved this problem sufficiently by special nature of the code entities. However, it is possible to decrease this problem also by additional services of the environment like triggers, priorities and FIFO data structures.

Another aspect of this communication concept is the form of the communicated messages, i.e. of the data what can be stored in the blocks. In most of applications it is sufficient to work with messages containing pure data buffer, i.e. the receiver has to know how to read the data. Another possibility it is to use an type set, i.e. the receiver has to know how to interpret the data only. Potentially, a communication language could be used, but the so-called ontology problem is omitted here, providing correct interpretation of the communicated data is up to designer of the system.

Interesting feature of our approach is that the message passing (and intruding consecutively) could be imperfect. Potentially the sender can write two different messages to the same block consecutively and the receiver omits the first one and gets only the second message. Though this loss can be minimized by the mentioned additional services of the environment, it is usually desirable for real-time operation and filtering out of transient stimuli.

6 Purely reactive agents

Now let us focus to the communicating code entities form. We have already mentioned that in our approach the receiver of a message written to a block by a sender does not need any stimulus to read it. There is only one solution, which can provide this: the receiver has to repeatedly poll the block looking for the message. Therefore our code part is arranged as reactive agent, a process which performs sense/select/act cycle endlessly. In this cycle it starts with perception of stimuli (reading of blocks), selection follows (computing of reaction to the stimuli), and it ends with action (realization of the selected reaction by writing or erasing blocks). (For systems working with physical devices, there are

transducers providing conversion of messages stored in some blocks to commands known to the devices and vice versa.) Moreover, several further restrictions are imposed on the selection because of reactivity requirement. It must be realized by a simple algorithm and no counterparts of sensed data expressed by a representation language are allowed. Due to lack of representation language no explicit goal is present, no planning is possible. Further important restriction is that the agent is purely reactive. It means it must not remember any data from previous sense/select/act cycle internally, it has no internal long-term memory. (Of course, it can use a block in environment to store information for later use, however, it is not guaranteed that it will find the data, because of intruding.)

There are really many restrictions imposed to the code entity. Under those restrictions it is certainly more difficult to express ideas by code. However, all the restrictions enables efficient intruding.

7 Architecture summary

Let us summarize the presented architecture. We have used multi-agent system built from purely reactive agents communicated indirectly through environment using its services as read a message from a block, write a message to a block, erase a message from a block and potentially further additional operations. This structure is built up using bottom-up incremental development, step by step according to the former decomposition of the system behaviour to individual activities. Within each step, several new agents and blocks are added to provide a new activity. The new agents can manipulate the blocks stated formerly and in this way they can control or coordinate agents implemented in the previous steps of development. Thus the new activity acts as an intruder to the older ones. Designer has to find such arrangement here, that activation of appropriate agent at appropriate time is guaranteed. Each design of the new activity has to be tested and corrected until no undesirable behaviour is detected. During this process, designer is constrained by the requirement of purely reactivity. On the other hand he gets a profit from intruding enabled by purely reactive agents which are totally opened for such influence. In each phase, he has a plan what to do next, but only a vague image how to achieve it. He can undergo many iterations to implement a new activity sufficiently. Therefore, we can say that he rather refines than designs the system.

8 Architecture testing

Although we have tried to present a deduction leading to the presented architecture, many of our choices seems to be obscure. It is because these choices have been adopted due to experiences with many variants of the architecture used for various purposes. Principally, we have used the presented architecture for three purposes.

- We have developed real-time monitoring critical-mission application for practice (it serves to monitor temperature on loops of nuclear reactor). Here we focused to test capability of today's computers to provide sufficient computational power, since demand of the presented architecture is due to reactive agents high. Further we tested capability of such system to undergo many modifications. (It is fact well-known in software engineering that practically no system can undergo never-ending modifications, though theoretically it is possible.) Finally configurability, robustness and recovering from errors have been evaluated. Since it was not artificial intelligence problem, we were able to compare the results to similar systems with traditional architecture.
- We have developed simulator of non-trivial insect behavior (making offspring of cerceris bupresticida). Here we focused to utilization of intruding to provide complicated behaviour containing many actions both in sequence and in parallel.
- At the time we are trying to develop system simulating part of mind to produce similar results to those achieved by probands in a simple psychological test. It is very interesting that the imperfectness of intruding can explain occurrence of some probands' fails.

All our projects are implemented under QNX4 platform, where real-time and uniform message passing are convenient for such architecture.

9 General purpose agents

Unlike subsumption architecture, due to software environment, our architecture provides named blocks instead wires. This is an important difference, because we can build agents working with parameterized reference, manipulating with groups of blocks with similar structured names, etc. Such agents do not differ significantly from the ordinary ones in their code, but can serve for general purposes.

10 Conclusion

We have presented a software architecture for building intelligent systems. It is intended to overcome lack of formal problem specification. It is based on

- incremental bottom-up development
- decomposition by activity
- intruding
- indirect communication through environment
- purely reactive agents

May be, our explanation seems to be a long sequence of very big steps, but it summarizes more than 6 years of our research. At the time we have a long experience with the presented architecture. We have tested several variants and we prefer the presented one.

Our approach is strongly based on excellent ideas of R. Brooks and is also influenced by work of M. Minsky [10]. Mind simulation by this architecture is similar also to transducers and modules from Fodor's theory of mind. It is question, if it is possible to simulate by this architecture only low level of mind or also more. Fodor's arguments for central systems existence stated in his theory outline that the architecture has very strong limits. However we think it could serve to build many interesting systems producing behaviour what we cannot achieve by any traditional architecture.

References

1. Brooks R. A.: A Robust Layered Control System for a Mobile Robot. IEEE Journal of Robotics and Automation, RA-2, (1986), 14–23
2. Brooks R. A.: A Robots that Walks: Emergent Behaviors from a Carefully Evolved Network. Neural Computation 1:2, Summer, (1989)
3. Brooks R. A.: Intelligence without representation. Artificial Intelligence 47, (1991), 139–159
4. Brooks R.: Building Brains for Bodies. (A. I. Memo No. 1439), MIT AI Lab, Cambridge, Mass., 1993
5. Ciancarini P.: Lectures on coordination models and languages. EASSS '99, (1999)
6. Doran J.: Distributed AI and its Applications. In: Advanced Topics in Artificial Intelligence (V. Marik, O. Stepankova, R. Trappl, eds.) Springer-Verlag, Berlin, 1992
7. Goodwin R.: Formalizing Properties of Agents. (Report CMU - CS - 93 - 159), Computer Science Department, Carnegie Mellon University, Pittsburgh, Pensylvania, 1993
8. Kelemen J.: Multiagent symbol systems and behavior-based robots. Applied Artificial Intelligence 7, (1993), 419–432
9. Knight K.: Are Many Reactive Agents Better Than a Few Deliberative Ones? In: Proc. IJCAI '93, Chambery, 1993, 132–137
10. Minsky M.: The Society of Mind. Simon&Schuster, New York, 1986.
11. Stein L. A.: Imagination and Situated Cognition. (A. I. Memo No. 1277), MIT AI Lab, Cambridge, Mass., 1991.