Zbornik radova 49. Konferencije za ETRAN, Budva, 5-10. juna 2005, tom I
Proc. 49th ETRAN Conference, Budva, June 5-10, 2005, Vol. I

A CSP-BASED TRAJECTORY FOR DESIGNING FORMALLY VERIFIED EMBEDDED CONTROL SOFTWARE*)

Dusko S. Jovanovic, Geert K. Liet, Jan F. Broenink

Twente Embedded Systems Initiative, Drebbel Institute for Mechatronics,
dept. of Control Engineering, Univeristy of Twente
P.O.Box 217, 7500AE Enschede, the Netherlands
Phone: +31 53 489 2788 Fax: +31 53 489 2223
e-mail: d.jovanovic@utwente.nl

Abstract – *This paper presents in a nutshell a procedure for producing formally verified concurrent software. The design paradigm provides means for translating block diagrammed models of systems from various problem domains in a graphical notation for process-oriented architectures. Briefly presented CASE tool allows code generation both for formal analysis of the models of software and code generation in a target implementation language. For formal analysis a high-quality commercial formal checker is used.*

1. INTRODUCTION

The term "software crisis" has been coined some thirty years ago by Dijkstra [1]. From that moment on, this crisis has never ceased [2] – it just has transformed as the abilities of computer hardware transformed, along with the expectations of the users. It is expected that electronic artificial intelligence gets embedded in virtually any domain of everyday physical activities. The emergent knowledge society is rooted in the ubiquitous proliferation of the computer-based surroundings.

Pervasive computing, as this phenomenon is termed by IBM, stems from dramatic advancements in the micro- and nanoelectronics, according to the Moore's law. In order to attain a full benefit of the revolutionary miniaturization and corresponding increase of computing power, the hardware progress has to be proportionally paced by the software technology. However, that is not the case: the demands for harnessing the available hardware power are not matched with the mastery of crafting adequate software solutions. The lost balance between the progress of hardware and software technology causes virtually all "hi-tech" projects to experience tremendous delays, breaking budgets and unreliability – symptoms of the software crisis. Under the market pressure, the picture worsens taking into account premature forcing total computerization of many safety-critical systems.

A paradigm shift in the software production is what both academic and industrial research tries to attain, since curbing the hunger for computation power is less likely to happen. Projections in the next 5 to 10 years further into the "information age" reveal high expectations of the technology. Ubiquitous networked computing nodes, also named "electronic dust" are shaping everyday environments into so-called smart surroundings, lending themselves for the infrastructure of "ambient intelligence". Smart surroundings are characterized by high topological reconfigurability, (wireless) ad-hoc networking, concurrency and customization.

These are the requirements posed to the information technology. But what is the sought paradigm shift in software engineering that may empower these gigantic-scale intelligent systems?

First of all, it should be observed that the state-of-the-art level of the software production is hardly to be termed "engineering", but the *development* at best, if not *craftsmanship* or *art* in many cases. In order to admit a creative discipline to an engineering, it has to have certain properties, as formally rigorous design, quantified quality assessment and predictability. The metrics of the software production are not widely established, quality of the software is not predictable at the design time and is mainly guaranteed by testing – but as Dijkstra famously observed, "program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence". In short, mathematical (formal) reasoning in the software development process is missing, quite opposite to recognized *engineering* disciplines, as civil engineering, avionics, control or mechanical design.

The fight against (the symptoms of) the software crisis (productivity and quality) has yielded numerous software development methodologies and tools, called CASE – Computer Aided Software Engineering – tools (represented as a group in the middle of Figure 1). They deal with modelling software application domains into software architectures expected to provide satisfactory computer programs. Often the application domains have their own CAD – Computer Aided Design – tools, which help engineering (physical or information) systems whose some parts are implemented in software.

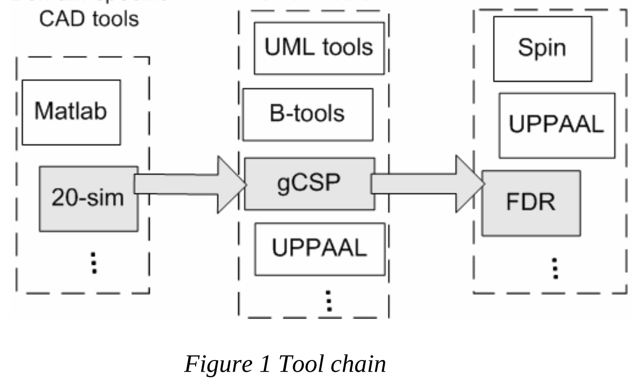

Figure 1 Tool chain

*) This research is supported by PROGRESS, the embedded system research program of the Dutch organization for Scientific Research, NWO, the Dutch Ministry of Economic Affairs an the Technology Foundation STW.

Particularly interesting for integration with the CASE tools are those CAD's able to produce source code for software-implemented components. Good examples are Matlab/Simulink and 20-sim for control applications (Figure 1).

However, using a CASE tool in software development process does not imply its formal underpinning. Moreover, predominantly used methodologies and tools in the software industry have no formal (mathematically rigour) background – as for instance the mainstream object-oriented UML paradigm. However, the research in formal methods for software engineering has not been less active (nor diverse) as inventing software development paradigms and tools. It resulted in numerous formal theories and notations and led to development of various formal checkers for behavioural analysis of descriptions in those notations. It is surprisingly disappointing that these two worlds have been existing in isolation, with just a few exceptions (as B-method and UPPAAL).

In the following two sections a background of the indicated tool chain in Figure 1 is clarified. Section 4 presents a example of practical significance. The contribution of the reported work is summarized in Section 5.

2. CSP/CT

CSP (Communicating Sequential Processes) [3] are one of four major concepts pertaining to a formal basis of concurrent programming. The three other concepts are: CCS (Calculus of Communicating Systems), Petri Nets and various algebraic models of parallel processes, with proof systems based on classical mathematical logic, and its extensions, eg. temporal logic and modal logic. Out of all four, CSP is the closest to parallel programming languages [4]. The concepts of CSP model of concurrency are regarded as one of the key influential contributions in computer science, while the book [3] is the third most-cited computer science reference. On the practical side, Ada synchronous concurrency model is CSP-based, while revolutionary transputer has been programmed by pure CSP implementation language occam. After transputer disappearance in mid 1990's, a few universities took on initiative to provide occam-like approach to programming concurrency in the form of libraries for mainstream languages. Compared to the other formal theories, the CSP has a substantial advantage in availability of powerful tools for model checking, among which is one of the most advanced commercial formal checker FDR [5], proven useful in verifying design of utterly complex embedded systems [6].

Control Laboratory at the University of Twente has a rich experience in implementing advanced robotic control strategies on transputer platforms. Within this group one of the most successful practical CSP-based set of libraries for Java, C and C++ called Communicating Threads (CT libraries) is developed and maintained [7, 8].

Recent developments are a graphical language for designing CSP-based process-oriented architectures [8] and a CASE tool (called gCSP) for creating, analysis and code generation of graphical models for CSP-based concurrent software [9].

```
datatype theType = some_val | another_val

channel ch1 : theType
channel ch2 : theType

Producer12 = ch1!some_val -> ch2!another_val ->
Producer12

Consumer12 = ch1?aVar -> ch2?bVar -> Consumer12

Consumer21 = ch2?bVar -> ch1?aVar -> Consumer21

SystemDC = Producer12 [|{| ch1, ch2 |}|] Consumer21

SystemDF = Producer12 [|{| ch1, ch2 |}|] Consumer12
```

3. CSPm, gCSP AND 20-SIM

This paper focuses on an automated trajectory of producing formally verified CSP-based concurrent software implemented by CT libraries. The trajectory comprises modelling of process-oriented architectures by CSP diagrams [8] in the gCSP tool [9], transformation of the graphical models into formal description scripts and automatic generation of the CT code. Actually, the graphical models are transformed also by the code generation means into formal description language readable by the model checker FDR, called machine-readable CSP (CSPm).

Since the research is carried out in the Control Engineering group, the pilot application domain are concurrent implementations of embedded control software systems. The design of control laws and strategies is accomplished by 20-sim, a powerful CAD tool for modelling and simulation of dynamic models, with special and comprehensive provisions for designing control laws.

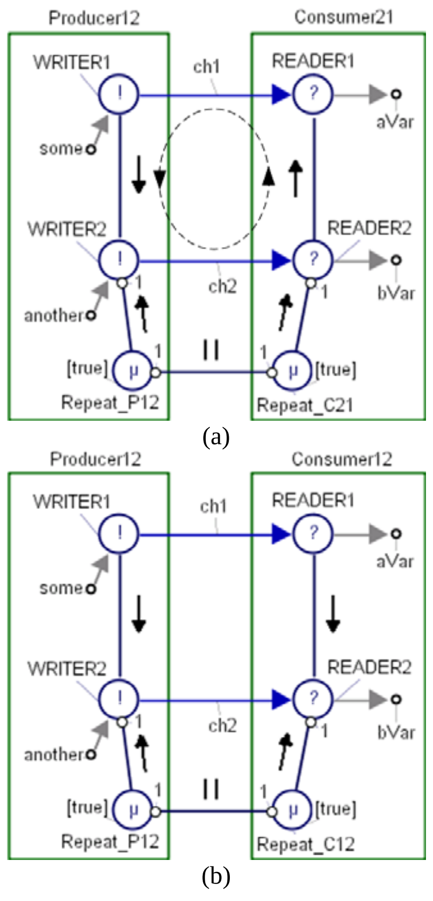

Figure 2 Deadlock condition and deadlock freedom

Especially interesting is the feature of 20-sim to automatically generate C code for complex dynamical models. This feature together with the gCSP's capability to automatically generate concurrent frameworks for (sequential) control code components additionally contributes to the reliability of the control software as a whole.

For illustrating the formal verification part of the whole design trajectory, detection of the most infamous problem in concurrent programming – deadlock – is used. For this a limited scope of the graphical language presented in Figure 2 suffices. For detailed information see [9]. The framed listing shows a corresponding example of CSPm script.

Communication patterns of processes `Producer12`, `Concumer12` and `Consumer21` (rectangles) are described by refining their internals with primitive (circled) writer processes for outputting data (CSP "`!`" operator) to synchronous channels (`ch1` and `ch2` in these examples) and inputting data from the channels (CSP "?" operator). Circled μ primitive processes describe repetitive execution of the processes (i.e. their internal compositions) – in these examples the repetitions are endless, which is indicated with the `true` iteration condition. Repetitive execution of a process or a composition of process is in the graphical language model by connected the subject of repetition with a μ primitive processes by a sequential relationship (while channels are arrowed lines, compositional relationships are also represented by lines, accompanied with a CSP operator symbol aside, which is an arrow for sequential composition). The bubble indexed with 1 denotes that the sequences of primitive writers/readers are repeated as a group. Likewise, bubbles on the parallel composition relationships (denoted by "`||`") place the entire internals of the top-level processes in parallel compositions.

In the script the type of specified channels (`ch1` and `ch2`) consists of two constant values (`some_val` and `another_val` respectively). In the script these two constants represent any value that may be contained by the variables `some` and `another` (see graphical representation of `Producer12` on Figures 2a) and 2b)). `Producer12` is defined by the following communication pattern: first value of the variable `some` is output to the channel `ch1`, and then value of the variable `another` is output to the channel `ch2`. This sequence of channel activations is diagrammed in Figures 2a) and 2b) by the sequential relationships. Similarly, sequences that describe how processes `Consumer12` and `Consumer21` communicate with their environment are consistently described in the CSPm script and the diagrams. The specification of the producer process combined with the consumer processes with the two different communication patterns (with respect to the order of channel activations) describes two different systems (processes), `SystemDC` and `SystemDF`. Figures 2a) and 2b) correspond to these to systems respectively. In the graphical representation of the coupling of `Producer12` and `Consumer21` one may notice a cycle - a closed path uniformly oriented by the direction of sequential relationships and synchronous channels (orientation of the channels does not matter). Analysis of the script by FDR indicates deadlock freedom of all processes except the process `SystemDC`, which suffers from the deadlock condition. The reason is simple: communication patterns of `Producer12` and `Consumer12` are compatible, while those of `Producer12` and `Consumer21` are not.

While `Producer` 12 tries first to output `some_val` on the channel `ch1` and then `another_val` to the channel `ch2`, `Consumer21` attempts reading from the channels in exactly opposite order (note the direction the upper sequential relationships). Therefore, on the attempted rendezvous synchronization processes wait on each other forever. A typical deadlock situation.

4. A PRACTICAL EXAMPLE

To illustrate a bit more realistic (and useful) context of

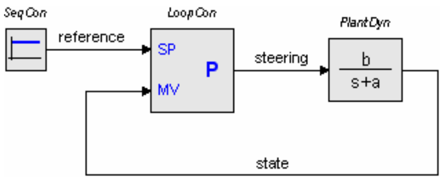

Figure 3 20-sim closed loop model

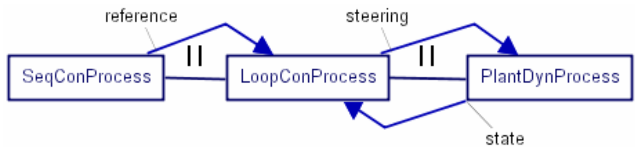

Figure 4 gCSP closed loop model

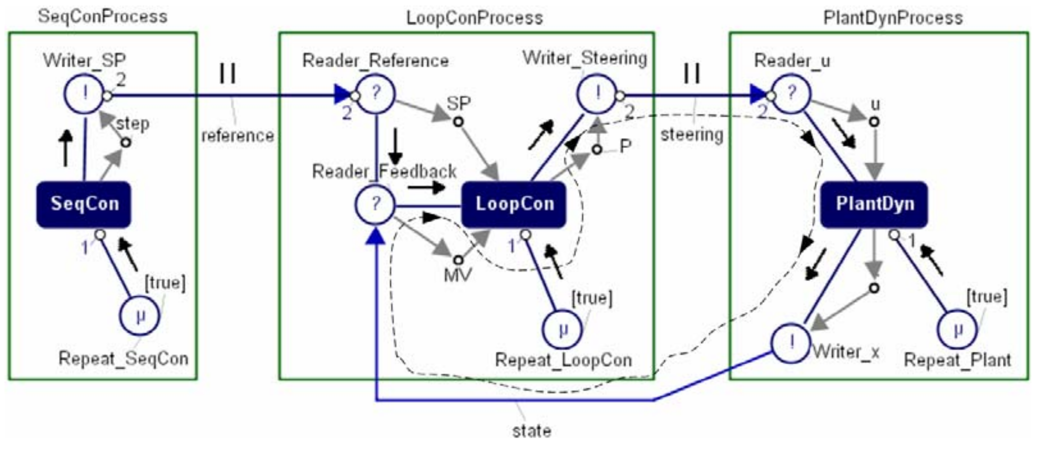

Figure 5 Model at a lower hierarchy level

using the outlined design trajectory, a block diagram of an basic closed loop system modelled in 20-sim is considered (Figure 3). Loop controller (`LoopCon`) on basis of the setpoint reference from a sequence controller (`SeqCon`) and feedback signal from a controlled object (`PlantDyn`) manages the steering values to the controlled object (plant). Translation to the CSP diagram in Figure 4 keeps the original topology by mapping functional blocks of 20-sim into gCSP processes and signals into channels.

Usually only controlling part of a 20-sim design gets implemented on the target platform; for the Hardware-in-the-Loop simulations, controlled object dynamics is taken from a control loop model and implemented. For the sake of clarity, in this example it is assumed that the complete dynamical model is refined towards implementation in order to verify soundness of the refinement concept. Internals of the processes from Figure 4 are present in Figure 5. Primitive readers and writers model inputting and outputting data in/from the processes. Data are manipulated by the algorithms represented by rounded rectangles (primitive code blocks). The μ processes denote repetitive nature of the data processing encapsulated by processes.

Execution order of the primitive processes is modelled by the sequential relationships. The top-level processes are however composed in parallel. The compositions of processes in Figure 5 reflect an intuitive arrangement: all data are first inputted in each sampling period, processed by the code blocks, and than outputted. This represents a straight forward mapping from the 20-sim structure. One should always keep in mind that the sophisticated 20-sim simulation engine treats a dynamic model as one (complex) set of equations. Possible causality conflicts (as algebraic loops) are solved by resorting the order of calculations by the simulation engine automatically. When this assumption is neglected, and the model is decomposed into elemental parts, the resulting composition may exhibit problems that were not seen during dynamical simulation in the CAD tool. Consequently, the parallel composition presented in this example suffers from a deadlock! Similarly to Figure 2a), a cycle of sequential relationships is present here as well. For the example in hand, formal verification in FDR that should precede program implementation and testing, would reveal the deadlock at the top compositional level, although all constitutive processes are deadlock-free.

There are many ways to resolve deadlock conditions. Due to the space limitation, the most simple, drawing from the elemental examples in Figure 2, will be applied in this situation. Although not applicable in most practical situations, due to this example simplicity, a simple reorganization in the order of activating operation in `PlantDynProcess` solves the problem. Namely, deadlock elimination amounts to breaking uniform orientation in the closed path in the model. That can be done by outputting the state of the dynamical plant prior inputting a new steering value. Simply, sequential relationship between the code block `PlantDyn` and primitive writer `WRITER_x` should be removed, and a new one connecting `WRITER_x` and `READER_u` established. Correctens of this solution (for this particular example) and elaboration of other solutions can be found in [9]. After CSPm code generation, FDR model checker verifies the deadlock freedom of the model. The model can be code generated in the CT library and successfully executed.

5. CONCLUSIONS

The paper presents principles of an automated trajectory for implementing formally verified software, continuing a long-term research in supporting structured development of embedded control systems [7, 10]. Therefore it starts with dynamical modelling of controlled object and control laws development. The concurrent software architecture inherits the topology of the original problem configuration. After software architecture refinement, its deadlock freedom is verified. Verified design can be then automatically translated to the operational code.

To our knowledge, this is the first tool for visualizing CSP-based designs. Moreover, the described method of using one model for both formal verification and automatic code generation is one of just few known to date.

The principles of formal verification were demonstrated on the deadlock prevention. The FDR model checker allows also other useful analyses, as determinism and livelock-freedom checks. The design refinement verifications (checking an implementation versus a system specification) would require an extension to the CSPm generator of the gCSP tools. Experimenting with these formal checks would be the first steps in further methodology improvements.

REFERENCES

[1] E. W. Dijkstra, “The Humble Programmer,” in *Communications of the ACM*, vol. 15, 1972, pp. 859-866.

[2] [W. W. Gibbs, “Software's Chronic Crisis,” *Scientific American*, 1994.

[3] C. A. R. Hoare, *Communicating Sequential Processes*: Prentice Hall, 1985.

[4] J. Olszewski, “CSP laboratory,” *ACM SIGCSE*, vol. 25, pp. 91-95, 1993.

[5] FormalSystems, “FDR2 Refinement checker for CSP models” http://www.fsel.com, 2004.

[6] G. H. Broadfoot and P. J. Hopcroft, “Combining the Box Structure Development Method and CSP”, *19th IEEE International Conference on Automated Software Engineering*. 2004.

[7] D. S. Jovanovic, G. H. Hilderink, and J. F. Broenink, “A design environment for developing and testing concurrent software for embedded control systems”, *XLVI Conference ETRAN*. Banja Vrucica, Bosnia and Herzegovina, 2002.

[8] G. H. Hilderink, *Manging Complexity of Control Software through Concurrency,* University of Twente, Netherlands, 2005.

[9] D. S. Jovanovic, *Designing dependable process-oriented software, a CSP approach,* University of Twente, Netherlands, 2005.

[10] D. Jovanovic, B. Orlic, and J. F. Broenink, “An automated transformation trajectory from a model of a controlling system to the control code”, *XLVII Conference ETRAN, XLVII Conference ETRAN*. Herceg Novi, Serbia and Montenegro, 2003.