Tools and Patterns for Dependable Concurrent Software

Dusko S. Jovanovic, *Student member, IEEE*, Jan F. Broenink, *Member, IEEE*

Abstract **– First we give reasons for choosing a process-oriented approach for building complex concurrent systems. Upon a brief review of dependability attributes of software-supported systems, means for increasing dependability in process-oriented architectures are illustrated.**

Keywords **— CSP, concurrent exception handling, dependability, design patterns and tools, formal analysis.**

I. Introduction

THE more software is implicated in all walks of real life, the more its structure has to reflect the nature of this real world. One of the characteristics of functioning of the physical world is *concurrency*. Therefore, concurrency in software is essential and cannot be ignored. There are numerous benefits of approaching software development with elicited concurrency [1]. However, it requires a proper handling to avoid pathological problems inherent to concurrent software (e.g. deadlock and livelock) and performance penalties (e.g. starvation and priority inversion). But these problems are just a small part in the general trouble of building *complex* software-supported systems.

The end of the 20th century witnessed a pronounced misbalance in public demands (in highly developed societies) and technological capabilities. On one side, it is expected that electronic artificial intelligence gets embedded in almost any domain of everyday physical activities; the emergent knowledge society is rooted in the ubiquitous proliferation of the computer-based surroundings. On the other side, virtually all software-intensive "hi-tech" projects experience tremendous delays, budget overruns and unreliability – symptoms of the everlasting software crisis [2]. Reports on project failures and disasters caused by software are overwhelming [3]. An obvious conclusion is that with proliferation of *pervasive computing* or so-called *ambient intelligence*, the modern society actually gets surrounded by an *unreliable*, and maybe worse, *unsafe* environment!

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 and the Technology Foundation STW.

dipl. ing. D. S. Jovanovic is with the dept. of Control Engineering, University of Twente, Netherlands (phone: 31-6-47830273; fax: 31-53-4892223; e-mail: djov@consultant.com).

dr. ir. J. F. Broenink is with the dept. of Control Engineering, University of Twente, Netherlands (phone: 31-53-4892793; fax: 31-53-4892223; e-mail: j.f.broenink@utwente.nl).

Research reported in this paper proposes a *concurrent* framework for complex software development not only because it is natural, but because it can be trustworthily *dependable*. Seemingly this may sound as a contradiction, since concurrent software is considered hard to understand, to analyze and hence to manage. We advocate that it is possible (and yet necessary) to *build software in a concurrent way and be certain about its dependability qualities*. This paper presents results of research inspired by finding *complementarities* of different dependability mechanisms and techniques for concurrent software tailored in a *process-oriented* fashion.

We define dependability according to [4]. Dependability is the ability to deliver service that can justifiably be trusted. It is the system property that integrates several vital software quality attributes (Fig. 1).

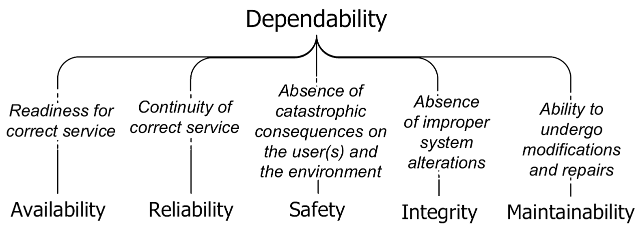
Dependability
Readiness for correct service
Continuity of correct service
Absence of catastrophic consequences on the user(s) and the environment
Absence of improper system alterations
Ability to undergo modifications and repairs
Availability
Reliability
Safety
Integrity
Maintainability

Fig. 1. Dependability attributes according to [4]

Focus of the reported research encompasses soft-ware development issues of all branches of dependability presented in Fig. 1. We do not consider the complementing concept of security [4] – our research deals with all *unintended* violations of a software system trustworthyness, thus not malicious attacks. It focuses on rectifying insufficiencies in development of concurrent software.

Our approach to building dependable concurrent software is *process orientation*. In Section II we describe what process orientation is and give it a formal background in a subset of the process algebra CSP. Section III highlights the ways in which software dependability in our framework is being increased. Concrete mechanisms and patterns are briefly presented in Section IV. The tool support (*gCSP*) and the implementation libraries (*CT*) are subject of Section V. Conclusions and the future work on our process-oriented framework for concurrent programming, referred to as *CSP/CT*, are summarized in Section VI.

II. Why Process Orientation and How

A common concurrency model is multithreading, which is the ability to have more than one task occurring in a program at the same time. Techniques for programming multithreading within the object-oriented paradigm rely on various constructs for synchronization scattered over objects and communication among objects. On single processor machines, only one thread can be executed at the time. The control flow goes from one object to another and as such is *not* object-oriented. Therefore, the bare use of multithreading increases complexity (i.e. hinders understanding) of concurrent programs.

The term *process orientation* pertains to a variant of the dataflow-driven software architecture paradigm. "A process-based architecture abstracts away from objects. Objects structure data and code while processes structure behaviour. Unlike objects, processes embrace observable properties of a concurrent program, such as reactivity, timeliness, responsiveness, priorities, and performance" [1]. Under *process-oriented* architectures we assume that the data processing algorithms of a program are confined within *processes* that exchange data via *channels.*

The vocabulary of processes and channels is the basis of the Communicating Sequential Processes (CSP) process algebra proposed in 1978 by Hoare [5] to address the most cumbersome problems of concurrent programming, as synchronisation primitives, nondeterminism etc. When based on CSP, channels (*communication relationships*) are synchronous, following the *rendezvous* principle; execution compositions among processes are ruled by CSP constructs, possibly represented as *compositional relationships* (see IV-*A*). On the practical side, Ada's synchronous concurrency model is CSP-based, while the transputer [6] has been programmed by the pure CSP implementation language occam [7].

Therefore, the process-oriented software development paradigm proposed in this research has a formal background in CSP. It is now a well thought theory of concurrent systems, applied by a few big software companies (as IBM [8], QinetiQ [9]) and supported by a sophisticated model checker, FDR [10].

After the transputer disappearance in mid 1990s, a few universities took the initiative to provide an occam-like approach to programming concurrency in the form of libraries for mainstream languages. Versions from the University of Twente, developed at the Control Engineering department, are called Communicating Threads (CT) – see Subsection V-*A*.

III. Dependability potentials of CSP/CT

When looking at interaction among software components in object-oriented designs, notably there is a rather liberal flow of information through and among objects. This is *not* a favourable property for high-integrity systems, where possible error propagation should be strictly confined. Process orientation intrinsically favours restricted "channelled" communication among software functional entities (processes), assuming arbitrary concurrency among them. Furthermore, the formal background of our process-oriented approach inherently offers a power of minimizing presence of certain architectural *development* errors through formal verification (FV) of properties as deadlock- and livelock-freedom.

However, run-time *transient* errors and *environmental* failures cannot be checked in advance. For covering *anticipated* errors in a robust application and its environment, exception handling mechanisms (EHM) are regarded the most important fault tolerance tools. Hospitality of the CSP/CT environment to dynamic concurrent exception handling is highlighted in Subsection IV-*D*.

For fighting *unanticipated permanent development* errors, static redundancy means are adequate. This paper presents a small set of selected industry-recognized static redundancy instruments in form of *design patterns* (DP) particularly suitable for the CSP/CT environment. Furthermore, for low-level implementation issues, trustworthiness of the elaborated design trajectory relies on automatic code generation (ACG), which is rooted in the supporting tools: code generation engines of the gCSP tool producing code compliant with the API of the CT libraries.

For a diagrammatic classification of different kinds of errors threatening dependability attributes of a software system and different means for the coverage see Fig. 2.

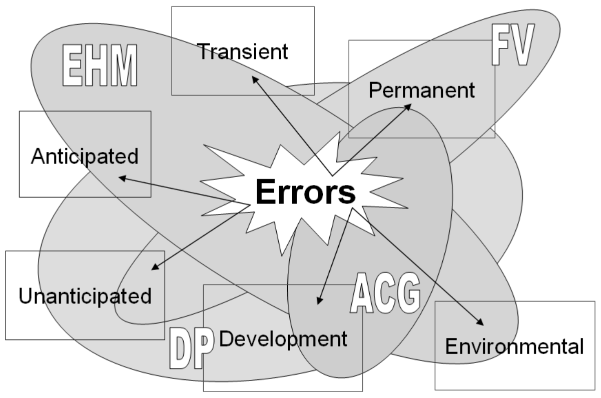

Fig. 2. Approaches to increase dependability:

FV	– Formal verification;
EHM	– Exception handling mechanism;
ACG	– Automatic code generation;
DP	– Design patterns

This classification of errors is far from exhaustive (for a more complete treatment see [4], [11]), though in the scope of this research it is useful for illustrating the approach to complementary means of increasing dependability of process-oriented architectures.

IV. Dependability patterns and mechanisms for CSP/CT

The concepts worked out in this research are briefly illustrated in this section. For an extensive elaboration the reader is referred to the corresponding chapters in [11].

A. Graphical modelling

A first quality aspect of a software design paradigm is the ability of *modelling* the designs. The modelling paradigm for the CSP-based concurrent software in form of a graphical language is elaborated in [1] as *CSP diagrams*. This research further extends the graphical language towards practical applications in control software. The language is formally underpinned by machine-readable form of CSP – *CSPm* [12], [11].

Fig. 3 shows basic CSP vocabulary expressed in the graphical language: processes connected by communication relationships (channels) – arrow headed lines.

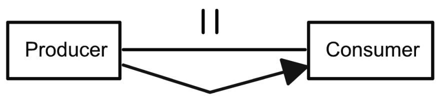

Fig. 3. Communicating parallel composed processes

Execution concurrency among processes is specified by compositions, ruled by CSP operators, in the CSP diagrams represented as compositional relationships; these lines are distinguished from channels as being undirected and adorned by an extended set of the CSP operators' symbols (Table 1).

TABLE 1. COMPOSITIONAL CONSTRUCTS IN CSP/CT

$\rightarrow$ sequential	$\overrightarrow{\triangle}$ exception
$\|\|$ parallel	$\overleftarrow{\|\|}$ priparallel
$\square$ alternative	$\overleftarrow{\square}$ prialternative

B. *Automatic code generation (ACG)*

There are two principal reasons for making a software modelling tool capable of generating source code out of (graphical) software models:

1. Elimination of manual transformation from modelled behaviour to the low-level code. The manual coding of abstract models is proven to be lengthy (i.e. expensive) and error-prone.
2. Although it is claimed that "software does not deteriorate with use", it is also true that the structure of the software actually degenerates with maintenance ("software aging", [13]). A carefully designed code generation engine generates well structured programs regardless how complex the software systems grow.

Having these qualities in mind, it is clear that automatic code generation mechanisms are effective means in covering a class of *implementation development* errors.

C. *Formal verification (FV)*

Benefits from formal analysis of models of (concurrent) systems are well known; while the testing procedures – doesn't matter how carefully designed – intentionally target only expected sources of malfunctioning (and only accidentally reveal other design insufficiencies), an exhaustive checking is possible only by deployment of formal verification.

Formal verification for CSP/CT is also achieved through automatic code generation – the graphical designs are transformed into the CSPm formal specifications. In [14] checks against deadlock conditions as a form of architectural *permanent development* errors are elaborated and demonstrated. However, on basis of the same CSPm specifications, checks on properties of livelock freedom, safety and determinism are straightforward.

D. *Exception handling mechanism (EHM)*

Exception handling is a *dynamic redundancy* mechanism that allows system architects to distribute dedicated corrective or alternative code components at appropriate places within the software structure to maximize effectiveness of error recovery. Therefore, it successfully covers a broad class of *anticipated transient* errors and effects of *environmental* failures within a software system. The most important feature expected from an EHM is separation of *ordinary* execution code and part of the code for treating *exceptional* situations. The other *eleven* relevant properties that an EHM should fulfil are quoted in [15].

In the CSP/CT framework a concurrent EHM benefits from separation of software components' concerns in well-defined processes; the important aspect of exception propagation is excellently captured by the CSP (occam) hierarchical structure.

E. *Design patterns (DP)*

For covering *unanticipated development* errors in a software design one has to reach for *static redundancy* – redundant components that remain in use whether or not any errors occur. In this research a few selected design patterns are tailored for process-oriented applicability. Suitability for the process orientation was one selection criterion; the other two were non-obtrusiveness to the original (error-unaware) design and wide recognition in industrial practice.

N-version software replicas bring in redundant algorithms derived from the same functional specification and developed by different tools and/or teams. Having an odd number (*N*) of replicas allows for majority voting policy if there is a disagreement in outcomes from different software component versions. In CSP networks a critical process is being replaced by an error-tolerant one that contains multiple processes with the same functionality as the original and coordination processes. In that way the original structure stays unchanged. Moreover, issues of synchronisation among different versions are elegantly addressed by the CSP/CT constructs.

Logging and optionally *monitoring* traffic on the CSP channels is another way to increase confidence in a proper functioning. Since a CSP process' behaviour is defined only through its interaction via interconnecting channels, having a comprehensive overview about the software functioning is possible *without* changing the functionality of the processes, but just by using "monitoring-aware" (probe) channels. Monitoring can be active, which means that corrective actions can be taken, for instance using the exception handling mechanism.

Watchdogs take care about major disturbances of a CT network activity on basis of temporal behaviour disruptions. Also, it is possible to monitor slack CPU time and remaining unallocated memory in order to give early warnings on suspicious trends in run-time.

V. TOOLS

A. *CT libraries*

Three versions of the CT libraries – for Java, C and C++ [1], [16] – represent a process-oriented implementation layer for CSP/CT concepts in practical applications. The API of the libraries offers the CSP/occam-like vocabulary: processes, compositional constructs and channels. The set of the operators from CSP is extended with prioritized versions for parallel and alternative constructs (following the occam extension for priorities, reflected by the prioritised constructs in Table 1); the concept of exception handling has yielded the exception construct.

Recent extensions to the CT libraries are pre-programmed design patterns (DP) facilities as *probe channels* for logging and monitoring, watchdog components at several levels (from scheduler- to user-level processes) and the EHM facilities.

B. *gCSP CASE tool*

In order to allow modelling CSP/CT designs by CSP diagrams, automatically generate code out of them, so prove the concept of formal verification and facilitate applications of dependability design patterns, at the Control Engineering a CASE tool named graphical CSP (*gCSP*) is developed [17].

Besides editing hybrid CSP diagrams (with communication and composition relationships like in Fig. 3), separating communication and composition views is possible as well. The other crucial functionality of the tool is code generation. The major problem to overcome in automatic code generation is the conceptual distance between highly abstract graphical models and the target programming language; spanning distant system specifications is often done by multiple transformations through gradual refinement stages [18]. As an interdomain towards the generated code, the tool presents the CSP/CT architectural hierarchies by a compositional tree, called the C-tree (Fig. 4). The tree view is also particularly useful for navigation through complex designs, since it ideally reflects the CSP/occam hierarchical network structure.

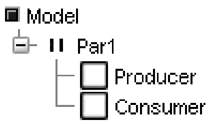

Fig. 4. C-tree corresponding to Fig. 3

The mechanism of code generation in gCSP is used to address the aforementioned FV and ACG error coverage aspects. These goals are supported by code generators for the CSPm formal descriptions and the implementation code for C++ version of the CT library. Producing one or the other machine-readable presentation *does not require any* change in a gCSP model. The two transformations are produced from the single model, by choosing different code generator engines of gCSP. A generated CSPm formal specification of a graphical design is directly liable to model checking with the high-end model checker FDR. Implementation code is generated ready for compilation and deployment. Bespoke algorithms, hardware drivers manipulation as well as some debugging facilities are also defined within the model. This means that manual interventions upon implementation deliverables are eliminated.

gCSP proves useful to support adding EHM and DP protective measures in form of layers on the graphical designs. Both mechanisms are superimposed on an existing CSP/CT network modelled in the tool.

VI. Conclusions

We claim covering the most pronounced sources of errors in concurrent software through combination of various complementary dependability means. Since the research is carried out in the Control Engineering group, the pilot application domain are concurrent realizations of embedded control software systems [1], [11].

Currently the extensions of the framework advance in a few directions. One of them is a simulation support at the level of the CTC++ library that would allow predictions of temporal behaviour of a CSP/CT design. Other two promising avenues to similar goals are timing analysis based on the operational CSP semantics combined with theory of timed automata and executable specifications (design animations) within the gCSP tool. Lastly, Table 1 tends to be extended with at least one more compositional operator, namely for modelling the watchdog protective layer. Compositional extendibility surely does belong to this software design paradigm.

References

[1] G. H. Hilderink, "Managing Complexity of Control Software through Concurrency," PhD thesis, University of Twente, NL, 2005.

[2] W. W. Gibbs, "Software's Chronic Crisis," Scientific American, 1994.

[3] N. G. Leveson, Safeware: System Safety and Computers, Addison-Wesley, 1995.

[4] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," IEEE Trans. on Dependable and Secure Computing, vol. 1, pp. 11-33, 2004.

[5] C. A. R. Hoare, "Communicating Sequential Processes," Communications of the ACM, vol. 21, pp. 666-677, 1978.

[6] R. Ivimey-Cook, "Legacy of the Transputer," presented at WoTUG22: Architectures, Languages and Techniques for Concurrent Programming, University of Keele, UK, 1999.

[7] INMOS, occam 2 Reference Manual, Prentice Hall, 1988.

[8] J. Lawrence, "Practical Application of CSP and FDR to Software Design," in LNCS 3525, Springer-Verlag, pp. 151-174, 2005.

[9] S. Creese, "Industrial-Strenght CSP: Opportunities and Challenges in Model-Checking," in LNCS 3525, Springer-Verlag, p. 292, 2005.

[10] FormalSystems, "FDR2 Refinement checker for CSP models," 2004.

[11] D. S. Jovanovic, "Designing dependable process-oriented software, a CSP approach," PhD thesis, University of Twente, NL, to appear in 2005.

[12] J. B. Scattergood, "Tools for CSP and Timed CSP," D.Phil thesis, Oxford University, UK, 1997.

[13] D. L. Parnas, "Software aging," presented at International Conference on Software Engineering, Sorrento, Italy, 1994.

[14] D. S. Jovanovic, G. K. Liet, and J. F. Broenink, "A CSP-based trajectory for designing formally verified embedded control software," presented at 49th conference ETRAN, Budva, Montenegro, 2005.

[15] D. S. Jovanovic, B. Orlic, and J. F. Broenink, "On issues of constructing an exception handling mechanism for CSP-based process-oriented concurrent software," presented at Communicating Process Architectures CPA 2005, Eindhoven, NL, 2005.

[16] B. Orlic and J. F. Broenink, "Real-time and fault tolerance in distributed control software," presented at Communicating Process Architectures CPA 2003, Enschede, NL, 2003.

[17] D. S. Jovanovic, B. Orlic, G. K. Liet, and J. F. Broenink, "gCSP: A Graphical Tool for Designing CSP Systems," presented at Communicating Process Architectures CPA 2004, Oxford, UK, 2004.

[18] D. Milićev, (in Serbian) "Automatic transformation of models in software tools for modelling," PhD thesis, University of Belgrade, Serbia, 2001.