QUALITY INCREMENT OF PROCESS-ORIENTED SOFTWARE ARCHITECTURES BY CLASSICAL DEPENDABILITY DESIGN PATTERNS

Dusko S. Jovanovic, R&D embedded software, Neopost Technologies B.V., Netherlands
d.jovanovic@djov.net

Abstract – In a spectrum of instruments for increasing dependability of software systems, solid (permanent) development errors are being fought against by statically redundant protective components. This paper presents application of a couple of the most successful practical design patterns adapted for an embedded concurrent software design environment based on formal grounds of the CSP process algebra.

1. INTRODUCTION: QUALITY OF SOFTWARE

By talking to different stakeholders in a software project, one may see “software quality” recognized in different aspects: intuitiveness, simplicity, security, safety, performance, reliability, predictability, extendibility, price, documentation etc. Research reported in this paper assumes dependability as a crucial aspect of software quality. Dependability is in [1] defined as “ability to deliver service that can justifiably be trusted” or in other words as “ability to avoid service failures that are more frequent and more severe than is acceptable”. It is the system property that integrates several vital software quality attributes (Fig 1).

Fig 1. Dependability attributes according to [1]

This paper concentrates on one segment of a research into dependability of software embedded in reactive systems [2]. The context of this research is an approach to building dependable concurrent software for control applications whose quality draws from combining several established dependability instruments, as automatic code generation, formal verification, exception handling and applications of dependability design patterns. The aspects of automatic code generation and formal verification are elaborated in [3]; the concurrent process-oriented exception handling mechanism in [4]. This paper elaborates on a few classical dependability design patterns — watchdogs, logging, monitoring and N-version programming — tailored for a hosting process-oriented software architecture. Complementarity of all these dependability instruments is presented in more detail along with an elaborate discussion on notions of software quality in [5].

Section 2 of this paper introduces the hosting process-oriented framework named CSP/CT and its devices; moreover, it summarizes dependability potentials of this architecture and its comprehensiveness. Section 3 presents the applied dependability design patterns. Conclusions are listed in Section 4.

2. CSP/CT PROCESS-ORIENTED ENVIRONMENT AND ITS DEPENDABILITY STRENGTHS

The term process orientation [5] pertains to a variant of the dataflow-driven software architecting paradigms [6, 7]. Discussed environment offers structural entities that draw from the vocabulary of dataflow-driven architecting, internally programmed in an object-oriented fashion. In this way, it unifies process-oriented modelling at the conceptual architectural level and object-oriented reasoning for implementation reusability. “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” [8]. Under a process-oriented architecture we assume that the data processing algorithms of a program are confined within processes that exchange data (interact) exclusively via channels.

In turn, the vocabulary of processes and channels is the basis of the Communicating Sequential Processes (CSP) process algebra proposed in 1978 by Hoare [9] 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 Fig. 2).

On the practical side, Ada’s synchronous concurrency model is CSP-based, while the transputer [10] has been programmed by the pure CSP implementation language occam [11]. Set of compositional constructs used in CSP/CT is collected in Table 1. Language occam extends the set of three basic CSP operators (parallel, sequential, alternative) with prioritized variants for parallel and alternative. Exception and watchdog constructs are contributions of the dependability extensions [5, 8].

Therefore, the process-oriented software development paradigm proposed in this research has a formal background in CSP, which is now a well thought theory of concurrent systems, applied by a few big software companies (as IBM

Table 1. CSP/CT compositional constructs

<table>
<thead>
<tr>
<th>Construct</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>parallel</td>
<td>$\parallel$</td>
</tr>
<tr>
<td>alternative</td>
<td>$\triangleright$</td>
</tr>
<tr>
<td>sequential</td>
<td>$\rightarrow$</td>
</tr>
<tr>
<td>watchdog</td>
<td>$\mathcal{W}$</td>
</tr>
<tr>
<td>exception</td>
<td>$\triangleleft$</td>
</tr>
</tbody>
</table>

Fig. 2. Communicating parallel composed processes (compositional relationship denoted with compositional construct symbol; communication relationship – channel – with an arrow)
failures cannot be checked in advance. For covering architectural inheren- tly offers a power of minimizing presence of certain design trajectory relies on automatic code generation (ACG).

Drawing from its CSP conceptual roots and the CT implementation realization, discussed process-oriented framework is called CSP/CT. With the CT libraries empower the implementation level of the CSP/CT environment, CSP modeling and corresponding graphical language [8] are supported by a developed CASE tool, named gCSP [5]. The gCSP tool supports graph view to the CSP/CT model (Fig. 2) and a tree view (Fig. 3). Outside graphical modelling, the other essential feature of the gCSP tool is its code generation capability. The tool generates code compliant with the CT libraries, ocaml and machine-readable CSP – CSPm – [16], which is used for formal verification of graphical models.

![Fig. 3. The tree view corresponding to Fig. 2](image)

Main dependability advantages of the process-oriented ocaml-like software implementation environment is its formal foundation and safe communication model. By 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 between software functional entities (processes), assuming arbitrary concurrency among them.

![Fig. 4. Approaches to increase dependability: FV – Formal verification, EHM – Exception handling mechanism, ACG – Automatic code generation, DP – Design patterns](image)

The formal background of the CSP/CT framework inherently offers a power of minimizing presence of certain architectural development errors through formal verification (FV) of properties as deadlock- and livelock-freedom. For low-level implementation issues, trustworthiness of the CSP/CT design trajectory relies on automatic code generation (ACG).

However, run-time intermittent errors and environmental failures cannot be checked in advance. For covering anticipated errors in a robust application and its environment, CSP/CT offers a concurrent exception handling mechanism (EHM).

For fighting unanticipated solid (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.

For a diagrammatic classification of different kinds of errors threatening dependability attributes of a software system and different means for the coverage see Fig. 4. Precise definitions of the error classes used in this text are to be found in [1] or [5].

## 3. CSP/CT DESIGN PATTERNS

A design pattern is a generalized solution to a commonly occurring problem. Software design patterns result from proven maturity of certain concepts for building software. Much functionality is repeatedly required by various software applications – while mechanisms for solving them tremendously differ in flexibility, genericness, scalability and extensibility. Over the years, certain successfully applied mechanisms converged to versions with well-balanced properties.

Dependability design patterns are statically redundant, because of employing components that stay in operation regardless errors occur or not. In case of N-version programming, redundancy is literally reflected in existence of multiple parallel-running components of the same functionality. Various design patterns are devoted to observing a system in order to indicate and/or record suspect trends that may endanger system’s integrity. While general monitoring components observe states (conditions, variables) by value, watchdogs are established as specialized for monitoring temporal disruptions in a (sub)system’s functions. Many systems make use of logs and audit trails to perform post mortem analysis in order to improve system’s dependability by learning from ensued incidents or simply optimize the systems’ functionality or trace transient malfunction that do not leave material evidence. However, a system’s logs are useful both for on-line as well as off-line analyses.

### A. CSP/CT watchdogs

The software-liveness-check principle of watchdogs as special entities (hardware- as well as software-wise) that are regularly notified from a concerned software piece is as old as programmable electronics [17]. If a software component does not hit an assigned watchdog, it is considered malfunctioning. Classical watchdog circuit usually performs a (sub)system reset upon running up the watchdog timeout.

However, this classical drastic solution of reinitializing a system is often unacceptable in safety-critical systems. CSP/CT watchdog mechanisms offer choice of various intervention levels. Another achievement is fitting this very well known dependability approach in the process-oriented framework of reasoning and graphical CSP modelling, yet in a way that an initial error-unaware architecture is not overwhelmed by the superimposed watchdog functionality. A third point of attention has been separation of the watchdog working space of the software that it looks after, so that hanging of the observed system does not cause failing of the watchdog as well.

### Table 2. Watchdog operation symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="set" /></td>
<td>set operation</td>
</tr>
<tr>
<td><img src="image" alt="hit" /></td>
<td>hit operation</td>
</tr>
<tr>
<td><img src="image" alt="remove" /></td>
<td>remove operation</td>
</tr>
</tbody>
</table>

135
Table 2 shows the three watchdog operations. Setting (and by that starting) a liveness watchdog is done before entering the body of repetitive operations. The designer may choose to hit the watchdog before engaging in any suspected operation – by inserting the hit watchdog operation in sequence with risky processes. However, more likely is a minimalistic approach, like in example in Fig. 5: placing only one watchdog hit in a repetitive algorithm (star-process in gCSP denotes repetitions). After completion of the repetitive set of operations the watchdog should be disengaged (which is modelled by the watchdog remove symbol). Setting, hitting and removing operations can be placed at the same compositional hierarchy level, or, as in this example, be distributed at different levels of a program. Care should be taken that every setting of a watchdog must be matched by a corresponding removal.

Fig. 5. Modelling use of watchdog facilities

Another watchdog pattern that builds on a simplification of the presented liveness watchdog is used for indicating deadline breaches by particular processes. This pattern would consist only of Fig. 5a) part: if a watchdog is set and removed in each sampling period (thus not hit within the parallel composition) – each watchdog timeout (which then must be equal to the desired deadline-period) run-up signifies breach of the deadline.

In case of exceeding the watchdog timeout, depending on the deployed intervention level, a special process (WDtimeoutHandling in Fig. 5) can be programmed to take control and drive the system to a safe state. The gCSP tool introduces a special design layer for all watchdogging elements, such that by simple manipulation with graphical layers, the watchdogging functionality can become transparent for designers that want to concentrate on other aspects of a design at hand.

B. CSP/CT logging and monitoring

For a process-oriented architecture rooted in CSP principles, where the fundamental building blocks are processes with behaviour completely defined by the traces of events they communicate over channels, an event registration mechanism has a particular potential. Observing the software behaviour is possible by only knowing traffic on the channels. This holds a promise of having an additional degree of confidence in a CSP/CT network without intervening upon an initial architecture or modifying processes with stable implementations.

The logging/monitoring (L/M in the sequel) mechanism is actually an augmentation of the CSP/CT communication layer (Fig. 6). The principal responsibility of the logging coordinator is to attach time stamps to the data it receives from other components in a program and to store the record to a medium (RAM, permanent memory devices, network etc). Monitoring layer can associate multiple data samples gathered from channels in order to unveil suspect trends in a system behaviour and to consequently add proper indication in the log file (a passive role), or to intervene by correcting the channel communication instances or throwing exceptions.

Fig. 6. Architecture of the L/M layer within CSP/CT

In case of a neutral role of the monitoring coordination component (or its absence), the system facilitates classical logging mechanism. Unlike the CSP/CT synchronous communication layer, L/M layer transfers data asynchronously (no rendezvous blocking between the model-level channels and the L/M coordinators) and implicitly (no channels to carry the data instances). This is because additional channels would be non-functional elements polluting an initial design and there is no need for formal analysis of this unilateral mechanism. This choice spares unnecessary context switches for better performance.

To reflect the presence of the L/M functionality at the modelling level, a concept of probe channels is introduced. In order to observe the behaviour of a CSP/CT network, the channels from an initial design that carry interesting data are being upgraded to (marked as) probe channels (grey in Fig. 6). In this way the original topology remains intact and there are no changes in interfaces and implementation of processes.

C. CSP/CT N-version programming

Multiple (N) software component versions (functional replicas) bring in redundant algorithms derived from a same functional specification and developed independently, by different tools, languages, teams etc, [18]. If there is a disagreement in outcomes from different software component versions, an odd number \( N > 2 \) of the functional components allows elimination of error influences by majority voting policy. The practice proves [19] that a 3-version software is 5
to 9 times more reliable than a corresponding single-version high-quality design.

The most common configuration for N-version programming, with N=3, is shown in Fig. 7. The case with three versions is minimal for applying the majority voting principle. For mass-product applications N=3 is already an expensive overhead due to triplication of the development effort; for specific safety-critical applications, the number of versions can be much larger than three.

![Fig. 7. A typical 3-version redundant scheme in CSP/CT](image)

In principle, the N-version programming scheme can be implemented by any software development paradigm. What differs is to which extend is the designer freed from maintaining the execution protocol among the software components in a N-version programming scheme. The component that compares results of redundant software versions has to provide for different execution times of different versions. In a case of disagreement, besides deciding which result(s) to proclaim correct, the comparator should indicate a malfunctioning version. In some architectures the comparator component has also a responsibility to explicitly invoke (start) each of the versions and synchronise input data distribution from the server component.

Due to the lack of space, the elaboration on the benefits of the CSP/CT programming model for implementing the replied versions scheme will be restricted to listing the most important virtues. For an extensive discussion see [5].

1. With organizing execution of redundant software versions within a repetitive parallel construct, burden of explicit invocation and synchronised termination of the versions does not exist.
2. Thanks to CSP/CT synchronous communication model, synchronisation of the versions and server and comparator components (processes) is automatically catered by the channels.
3. Replacing a process with an N-version variant is facilitated within the gCSP tool by choosing the “N-version” augmentation option upon a tree view node. In that way the structure of an initial model stays the same: additional constructions are confined within the augmented process.

4. CONCLUSIONS

This paper presented application of a few selected classical static redundancy approaches enriched and adapted to the CSP/CT architecture. The selection criteria were:

- Suitability for a process-oriented architecture, in particular CSP/CT,
- Non-obtrusiveness (transparency) to an initial, error-unaware design—minimizing any increase in the model/system complexity,
- Simplicity, i.e. a wide recognition in industry.

The CSP/CT framework proves suitable for addressing certain dependability issues explicitly. In addition to its virtues in managing complexity, design portability, real-time execution support and transparent distributiveness [5, 8], it qualifies for an interesting option for architecting concurrent embedded software.

The strengths of the process-orientation are also obstacles to its wider use. It imposes a disciplined and restrictive way of design, which defers arriving to first prototypes—though much more dependable—for benefit of detailed modelling and verification. The resulting implementation is less efficient, both with respect to performance (due to sophisticated constructions) and memory footprint. However, the assumption of this research was that the execution platforms are becoming faster and richer, either just by higher efficiency in using Silicon or by hardware parallelisation.

REFERENCES