On the granularity of components∗

Ákos Dávida, László Kozmab, Tamás Pozsgaić

adepartment of Mathematics and Computing, University of Pannonia
e-mail: davida@almos.uni-pannon.hu, pozsgai@szt.vein.hu

bDepartment of Software Technology and Methodology,
Eötvös Loránd University of Sciences
e-mail: kozma@ludens.elte.hu

Abstract

It is essential nowadays to be able to produce fault-free software products within a given time for an estimated cost. Component-based Software Development (CBSD) is arguably one of the best choices, though the isolation of components can also result in a more difficult integration process at the end of the development. In this paper we study how the verification of a component-based software system is influenced by changes in the granularity of the components it consists of. These early design decisions affect the complexity and flexibility of the composing objects of a system. The results focusing on the differences are illustrated by simple examples.

Keywords: component-based software development, granularity, synthesis of programs, model checking

1. Introduction

Defining the range of component granularity can be quite difficult because there are several factors to be considered: the level of abstraction, the likelihood of change and the complexity of a component, etc. In the case of a system consisting of too small components there may be consequences: the interaction between smaller components requires more time and resources. On the other hand, a component that is too large provides more complex interfaces, is subject to more frequent changes and makes a system using it less flexible. For these reasons it is essential for a system architect to find a balance between the factors of cohesion and coupling [8].

In the future we become more and more dependent of the proper functioning of computer systems, so new methods are needed to increase our confidence in the correctness of such systems. Today’s most widely used techniques of testing are

∗This research work was supported by GVOP-3.2.2-2004-07-005/3.0.
only able to post-verify applications and they are not able to guarantee that there 
are no more hidden errors left in the design or in the code. However, with the 
emergence of formal methods, it became possible to create correct programs with 
respect to a given specification. Different types of synthesis methods and model 
checking are the most widespread formal techniques in practice today.

2. Granularity supported by synthesis methods

There are several different methods for synthesizing programs [2, 4, 5, 7]. In 
the following sections simplified variants of the Producer – Consumer Problem is 
used to show a method based on Classical Logic introduced by G. R. Andrews [1]. The general steps of this solution are the followings:

- Defining the problem
- Determining the skeleton of the solution
- Generating an abstract solution
- Implementing the atomic operations (beyond the scope of this paper)

In order to generate an abstract solution the guard conditions (based on Dijkstra’s 
Weakest Precondition Calculus) must be determined, in such a way that the in-
variant should hold before and after the atomic instructions [3]. In our examples 
the atomic instruction <\texttt{await} E \texttt{then} S> is used, meaning that the sequence 
of statements $S$ is not executed until the evaluation of the logical expression $E$ is 
true.

2.1. Producer – Consumer Problem (coarse)

The first variant of the Producer – Consumer Problem is the simplest one 
consisting of 1 producer, 1 consumer and a buffer with capacity 1. Let $pt$ be the 
fullness-pointer of the buffer $B$.

\textbf{Specification:}
\[ I_a : 0 \leq pt \leq 1 \]

\textbf{Skeleton of the solution:}
\begin{verbatim}
var pt : integer := 0; var B, a, aa : item;
# invariant $I_a$
Producer ::
do true then
<pt := pt + 1; B := a>
od
\end{verbatim}
Consumer ::
do true then
<aa := B; pt := pt − 1>;
od

The guard conditions described earlier are the followings:
\( B_p = wp(pt := pt + 1, I_a) \Rightarrow pt = 0 \)
\( B_c = wp(pt := pt − 1, I_a) \Rightarrow pt = 1 \)

Abstract solution (with guards):
# variables are just as before
# invariant \( I_a \)

Producer ::
do true then
Deposit: <await \( pt = 0 \) then \( pt := pt + 1; B := a \)>;
od

Consumer ::
do true then
Fetch: <await \( pt = 1 \) then \( aa := B; pt := pt − 1 \)>;
od

Analysis:
With such an invariant \( I_a \), the increase/decrease of the buffer cannot be divided into microinstructions, must be handled as atomic instructions just as in the example above. Let us take a look at the following solution:

Producer ::
do true then
Deposit: \(<pt := pt + 1>; B := a; \)
od

Consumer ::
do true then
Fetch: \(<pt := pt − 1>; aa := B; \)
od

Executing the Producer and the Consumer processes it is possible that the following sequence occurs.

\(<pt := pt + 1>; <pt := pt − 1>; aa := B; B := a; \)

It means that a data is attempted to be read from buffer \( B \) before it has actually been placed there. However, there are several correct solutions if we still want a finer structure as the skeleton of the solution.
2.2. Producer – Consumer Problem (fine)

Specification:
It is based on the previous solution but a finer structure is used. Let \( pt \) be the fullness-pointer of the buffer, \( in_p \) and \( in_c \) the indicators of the critical sections.

Invariant \( I_b \):
\[
0 \leq pt \leq 1 \land 0 \leq in_p \leq 1 \land 0 \leq in_c \leq 1 \land 0 \leq in_p + in_c \leq 1
\]

Abstract solution (with guards):
\[
\text{var } pt : \text{integer} := 0; \quad \text{var } B, a, aa : \text{item}; \quad \text{var } in_p, in_c : \text{boolean} := 0, 0;
\]
\[
\#	ext{ invariant } I_b
\]

Producer ::
\[
\text{do true then} \quad \# \text{ invariant } I_b
\]
Deposit: \(<\text{await } pt = 0 \land in_c = 0 \text{ then } in_p := 1; \quad pt := pt + 1; \quad B := a; \quad <\text{await true then } in_p := 0; >;
\]
\text{od}

Consumer ::
\[
\text{do true then} \quad \# \text{ invariant } I_b
\]
Fetch: \(<\text{await } pt = 1 \land in_p = 0 \text{ then } in_c := 1; \quad pt := pt - 1; \quad aa := B; \quad <\text{await true then } in_c := 0; >;
\]
\text{od}

Analysis:

The producing and consuming processes have (finer structure) more granularity in this case, they express it better that producing and consuming are executed in such a critical section preceded by an explicit prologue and followed by an explicit epilogue.

Another advantage of this solution is that it can be easily extended to a system with a buffer sized \( K > 1 \), more producers and consumers.

2.3. General extension of the Producer – Consumer Problem

Specification:
In this variant there are \( M \) producers, \( N \) consumers and a buffer with capacity \( K \geq 1 \). Let \( pt_i \) (the number of items deposited in the buffer), \( pt_o \) (the number of items fetched from the buffer) be pointers where \( (pt_i - pt_o) \) is the fullness pointer, \( in_p \) the number of producers in their critical section and \( in_c \) the number of consumers in their critical section.

Invariant \( I_c \):
\[
0 \leq pt_i - pt_o \leq K \land 0 \leq in_p \leq 1 \land 0 \leq in_c \leq 1 \land pt_i \geq 0 \land pt_o \geq 0
\]
Abstract solution (with guards):
\[
\begin{align*}
\text{var } & pt_i, pt_o : \text{integer } := 0, 0; \text{ var } B[1..K] \text{ of item;} \\
\text{var } & in_p, in_c : \text{integer } := 0, 0; \\
\# \text{ invariant } & I_c
\end{align*}
\]
Producer ::
\[
\begin{align*}
do & \text{true then } \# \text{ invariant } I_c \\
\text{Deposit: } & <\text{await } in_p = 0 \text{ then } in_p := in_p + 1>; \\
& <\text{await } pt_i - pt_o < K \text{ then } pt_i := pt_i + 1>; \\
& B[pt_i \mod K] := a; \\
& <\text{await } \text{true then } in_p := in_p - 1>; \\
\od
\end{align*}
\]
Consumer ::
\[
\begin{align*}
do & \text{true then } \# \text{ invariant } I_c \\
\text{Fetch: } & <\text{await } in_c = 0 \text{ then } in_c := in_c + 1>; \\
& <\text{await } pt_i - pt_o > 0 \text{ then } pt_o := pt_o + 1>; \\
& aa := B[pt_o \mod K]; \\
& <\text{await } \text{true then } in_c := in_c - 1>; \\
\od
\end{align*}
\]
Analysis:
This is a general solution. Producers have mutual exclusion among them. Consumers also have mutual exclusion among them. However, it is still possible that producing and consuming can be executed concurrently. That is why it is worth to divide the prologue into two atomic instructions.

It seems that the finer structure (more granularity) of the system makes it possible to increase concurrency.

Note: This solution also works correctly in the case of a buffer with capacity \( K = 1 \).

The coarser granularity has the advantage of expressing and illustrating specific properties of programs better. This is illustrated by the following example.

2.4. A variant of the Producer - Consumer Problem

In this case there are \( M \) producers, \( N \) consumers and a buffer with capacity \( K \geq 1 \). Two types of consumers \( C_1 \) and \( C_2 \) can be distinguished whether 1 or 2 products are consumed at once.

Specification:
Invariant \( I_d : 0 \leq pt \leq 2 \)

Abstract solution (with guards):
\[
\begin{align*}
\text{var } & pt : \text{integer } := 0; \text{ var } B[1..2] \text{ of item;} \text{ var } a, aa, aa_1, aa_2 : \text{item;} \\
\# \text{ invariant } & I_d
\end{align*}
\]
P ::
do true then
<await pt ⩽ 1 then pt := pt + 1; B[pt] := a>
od

C1 ::
do true then
<await pt ⩾ 1 then aa := B[pt]; pt := pt − 1>
od

C2 ::
do true then
<await pt = 2 then aa2 := B; pt := pt − 1; aa1 := B; pt := pt − 1>
od

Analysis:
This solution has a very coarse granularity like the first example. It is prohibited
to produce and consume concurrently, but it is possible to produce an item and
consume the very same item alternately. As a result of this process C2 is never able
to consume 2 items from the buffer. This property is known as starvation problem.
However, at this level of abstraction it is possible to solve it by modifying the guard
condition of process C1.

C1 ::
do true then
<await pt = 2 then aa := B[pt]; pt := pt − 1>
od

3. Model checking

Model checking, as opposed to synthesis methods, is an automatic technique
for verifying finite state concurrent systems. It is sufficient for the user to provide
a high level representation of the model and the specification to be checked, so no
verification expert is needed. The procedure normally uses an exhaustive search
of the space of a system to either terminate with the answer true, indicating that
the model satisfies the specification, or give a counterexample that may give an
important clue in finding subtle errors in complex systems. In some cases infinite
systems may be verified using model checking in combination with various abstrac-
tion and induction principles. The verification is ideally completely automatic. In
practice, human assistance is usually needed to analyze the results.

The main challenge in model checking is dealing with the state space explosion
problem. This problem occurs in systems with many components that can make
transitions in parallel. During the past ten years considerable progress has been
made in dealing with this problem [6].
The model and the specification of the general extension of the Producer – Consumer Problem described in Section 2.3 (implemented in NuSMV) can be found in Section 5.

4. Conclusions

In this paper we illustrated how the early design decisions on the granularity of components may have an impact on essential properties of a system (e.g. concurrency, freedom of starvation) and how to organize them to meet certain demands, usually in the form of logical expressions.

The examples above were created manually by a synthesis method suggested by G. R. Andrews [1]. Although the algorithm is quasi-automatic, it is usually not possible to synthesize large and complex software systems in the real business world.

However, there are model checking tools available nowadays that are able to verify automatically whether a system model is correct for a given specification. It may be more complicated to describe the model of a system in detail, but the process of verification requires less time and resources.

5. The general version of the Producer – Consumer Problem implemented in NuSMV

In this scenario the system consists of $M$ producers ($M = 2$), $N$ consumers ($N = 2$) and a buffer with capacity $K \geq 1$ ($K = 2$). Figure 1 shows the evaluation of the specification formulas that can be further analyzed. Concurrency, for example, between a producer and a consumer is checked with the following temporal logic formula:

\[
\text{SPEC} \ EF((\text{prod1.state = producing} \mid \text{prod2.state = producing}) \& (\text{cons1.state = consuming} \mid \text{cons2.state = consuming}))
\]

The entire model of the Producer – Consumer Problem with all of the necessary local and global specifications can be found here.

MODULE main

DEFINE

max := 2 ;

VAR

semaphore_prod : boolean ;
semaphore_cons : boolean ;
buffer : {0,1,2} ;
prod1 : process produce(semaphore_prod,buffer,max) ;
prod2 : process produce(semaphore_prod, buffer, max) ;
cons1 : process consume(semaphore_cons, buffer, max) ;
cons2 : process consume(semaphore_cons, buffer, max) ;

ASSIGN
init(semaphore_prod) := 1 ;
init(semaphore_cons) := 1 ;
init(buffer) := 0 ;

SPEC AG !((cons1.state = consuming) & (cons2.state = consuming))
SPEC EF((prod1.state = producing | prod2.state = producing) &
   (cons1.state = consuming | cons2.state = consuming))

MODULE produce(semaphore_prod, buffer, max)

VAR
   state : {idle, entering, producing, exiting} ;

ASSIGN
   init(state) := idle ;
   next(state) :=
      case
         (state = idle) & (buffer < max) : entering ;
         (state = entering) & (semaphore_prod) : producing ;
         state = producing : exiting ;
         state = exiting : idle ;
         1 : state ;
      esac ;
   next(semaphore_prod) :=
      case
         state = entering : 0 ;
         state = exiting : 1 ;
         1 : semaphore_prod ;
      esac ;
   next(buffer) :=
      case
         -- increasing the buffer pointer with case separation
         (state = producing) & (buffer = 0) : 1 ;
         (state = producing) & (buffer = 1) : 2 ;
         1 : buffer ;
      esac ;

FAIRNESS !(state = entering)
FAIRNESS !(state = producing)
On the granularity of components

FAIRNESS !(state = exiting)

SPEC AG(state = entering -> AF(state = producing))
SPEC AG(state = entering -> AF(state = idle))
SPEC AG(state = producing -> AF(state = idle))

MODULE consume(semaphore_cons, buffer, max)

VAR
  state : {idle, entering, consuming, exiting} ;

ASSIGN
  init(state) := idle ;
  next(state) :=
    case
      (state = idle) & (buffer > 0) : entering ;
      (state = entering) & (semaphore_cons) : consuming ;
      state = consuming : exiting ;
      state = exiting : idle ;
    1 : state ;
    esac ;
  next(semaphore_cons) :=
    case
      state = entering : 0 ;
      state = exiting : 1 ;
    1 : semaphore_cons ;
    esac ;
  next(buffer) :=
    case
      -- decreasing the buffer pointer with case separation
      (state = consuming) & (buffer = 2) : 1 ;
      (state = consuming) & (buffer = 1) : 0 ;
    1 : buffer ;
    esac ;

FAIRNESS !(state = entering)
FAIRNESS !(state = consuming)
FAIRNESS !(state = exiting)

SPEC AG(state = entering -> AF(state = consuming))
SPEC AG(state = entering -> AF(state = idle))
SPEC AG(state = consuming -> AF(state = idle))

FAIRNESS
running

```
- specification #G (state \(\rightarrow A\) state = producing) in prod1 is true
- specification #G (state \(\rightarrow A\) state = idle) in prod1 is true
- specification #G (state \(\rightarrow A\) state = consuming) in cons1 is true
- specification #G (state \(\rightarrow A\) state = consuming) in cons2 is true

Figure 1: Result screen of the model checking process
```

References


