Proceedings of the 7th International Conference on Applied Informatics
Eger, Hungary, January 28–31, 2007. Vol. 2. pp. 219–228.

On the granularity of components*

Ákos Dávid[a], László Kozma[b], Tamás Pozsgai[a]

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

[b]Department 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 invariant should hold before and after the atomic instructions [3]. In our examples the atomic instruction <**await** E **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.

Specification:
$I_a : 0 \leqslant pt \leqslant 1$

Skeleton of the solution:
var pt : integer := 0; **var** B, a, aa : item;
invariant I_a

Producer ::
do true then
<$pt := pt + 1$; $B := a$>;
od

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:
$0 \leqslant pt \leqslant 1 \wedge 0 \leqslant in_p \leqslant 1 \wedge 0 \leqslant in_c \leqslant 1 \wedge 0 \leqslant in_p + in_c \leqslant 1$

Abstract solution (with guards):
var pt : integer := 0; **var** B, a, aa : item; **var** in_p, in_c : boolean := 0, 0;
invariant I_b

Producer ::
do true then # invariant I_b
Deposit: <**await** $pt = 0 \wedge in_c = 0$ **then** $in_p := 1$; $pt := pt + 1$>; $B := a$;
<**await true then** $in_p := 0$>;
od

Consumer ::
do true then # invariant I_b
Fetch: <**await** $pt = 1 \wedge in_p = 0$ **then** $in_c := 1$; $pt := pt - 1$>; $aa := B$;
<**await true then** $in_c := 0$>;
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 $K > 1$. 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:
$0 \leqslant pt_i - pt_o \leqslant K \wedge 0 \leqslant in_p \leqslant 1 \wedge 0 \leqslant in_c \leqslant 1 \wedge pt_i \geqslant 0 \wedge pt_o \geqslant 0$

Abstract solution (with guards):
var pt_i, pt_o : integer := 0, 0; var $B[1..K]$ of item;
var in_p, in_c : integer := 0, 0;
invariant I_c

Producer ::
do true then # invariant I_c
Deposit: <**await** $in_p = 0$ **then** $in_p := in_p + 1$>;
<**await** $pt_i - pt_o < K$ **then** $pt_i := pt_i + 1$>;
$B[pt_i \ mod \ K] := a$;
<**await true then** $in_p := in_p - 1$>;
od

Consumer ::
do true then # invariant I_c
Fetch: <**await** $in_c = 0$ **then** $in_c := in_c + 1$>;
<**await** $pt_i - pt_o > 0$ **then** $pt_o := pt_o + 1$>;
$aa := B[pt_o \ mod \ K]$;
<**await true then** $in_c := in_c - 1$>;
od

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 $K \geqslant 1$. 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 : $0 \leqslant pt \leqslant 2$

Abstract solution (with guards):
var pt : integer := 0; **var** $B[1..2]$ of item; **var** a, aa, aa_1, aa_2 : item;
invariant I_d

P ::
do true then
<**await** $pt \leqslant 1$ **then** $pt := pt + 1$; $B[pt] := a$>;
od

C_1 ::
do true then
<**await** $pt \geqslant 1$ **then** $aa := B[pt]$; $pt := pt - 1$>;
od

C_2 ::
do true then
<**await** $pt = 2$ **then** $aa_2 := B$; pt := $pt - 1$; $aa_1 := B$; $pt := pt - 1$>;
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 C_2 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 C_1.

C_1 ::
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 abstraction 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 $K \geqslant 1$ ($K = 2$). 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:

```
SPEC EF((prod1.state = producing | prod2.state = producing) &
       (cons1.state = consuming | 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)
```

```
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 AG (state = entering -> AF state = producing) IN prod1 is true
-- specification AG (state = entering -> AF state = idle) IN prod1 is true
-- specification AG (state = producing -> AF state = idle) IN prod1 is true
-- specification AG (state = entering -> AF state = producing) IN prod2 is true
-- specification AG (state = entering -> AF state = idle) IN prod2 is true
-- specification AG (state = producing -> AF state = idle) IN prod2 is true
-- specification AG (state = entering -> AF state = consuming) IN cons1 is true
-- specification AG (state = entering -> AF state = idle) IN cons1 is true
-- specification AG (state = consuming -> AF state = idle) IN cons1 is true
-- specification AG (state = entering -> AF state = consuming) IN cons2 is true
-- specification AG (state = entering -> AF state = idle) IN cons2 is true
-- specification AG (state = consuming -> AF state = idle) IN cons2 is true
-- specification AG !(cons1.state = consuming & cons2.state = consuming)  is true
-- specification EF ((prod1.state = producing | prod2.state = producing) & (cons1.state = consuming | cons2.state = cons
uming))  is true
```

Figure 1: Result screen of the model checking process

References

[1] Andrews, G. R., A Method for Solving Synchronization Problems, *Science of Computer Programming* 13 (1989/90), 1–21.

[2] Attie, P. C., Emerson, E. A., Synthesis of concurrent programs for an atomic read/write model of computation, *ACM TOPLAS*, 23 (2), (2001), 187–242.

[3] Dijkstra, E. W., A Discipline of Programming, *Prentice Hall*, (1976).

[4] Emerson, E. A., Clarke E. M., Using branching time temporal logic to synthesize synchronization skeletons, *Science of Computer Programming*, 2, (1982), 241–266.

[5] Hajdara, Sz., Kozma, L., Ugron, B., Synthesis of a System Composed by Many Similar Objects, *Annales Univ. Sci. Budapest.*, Sect. Comp. 22, (2003), 127–150.

[6] Krishnamurthi, S., Fisler, K., Foundations of Incremental Aspect Model Checking, *ACM TOSEM*, Vol. 16 (April 2007), 1–39.

[7] Manna, Z., Wolper, P., Synthesis of Communicating Processes from Temporal Logic specifications, *ACM TOPLAS*, 6, (1984), 68–93.

[8] Vitharana, P., Risks and Challenges of Component-based Software Development, *Communications of the ACM*, Vol. 46 (August 2003), 67–72.