Verified Component-based Software in SPARK: Experimental Results for a Missile Guidance System

Kung-Kiu Lau and Zheng Wang
School of Computer Science, The University of Manchester
Oxford Road, Manchester M13 9PL, United Kingdom
kung-kiu,zw@cs.man.ac.uk

ABSTRACT

SPARK is useful for developing reliable software for safety-critical systems, using the 'correctness-by-construction' approach. It also has verification tools that can be used to produce verified software. To tackle larger-scale development of verified software, components are useful. In this paper we show how to define and implement software components in SPARK and use existing SPARK tools to produce verified component-based software. We demonstrate our approach on a missile guidance system.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Correctness proofs*

General Terms

Verification

Keywords

Proof reuse, SPARK, software components, verified software

1. INTRODUCTION

SPARK [2] is the basis of the 'correctness-by-construction' approach for developing reliable software for high-integrity systems [6]. Correctness by construction means that defects are avoided or detected and removed as early as possible in the development cycle. This reduces production cost directly, as well as subsequent maintenance cost indirectly [1]. The approach has been demonstrated successfully on real-world applications, e.g. [9].

SPARK also has verification tools. With these tools, the correct-by-construction approach can produce fully verified software. In this paper, we show that we can develop verified *component-based* software in SPARK. Moreover we believe that this approach can scale up, and thus has the potential to address the scale problem posed by the Grand Challenge in Verified Software [11, 12].

Our belief is based on our view that the scale problem can only be tackled by a component-based approach, where software components are systematically composed into a complete system. We have formulated such a component-based approach, and we have implemented it in SPARK. This implementation enables us to use the SPARK tools to produce verified component-based software. In this paper, we explain our component-based approach and show how we implement it in SPARK. We illustrate our SPARK-based approach to verified component-based software with an industrial-strength case study for a missile guidance system.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGAda'07, November 4–9, 2007, Fairfax, Virginia, USA.
Copyright 2007 ACM 978-1-59593-876-3/07/0011 ...$5.00.

2. SOFTWARE COMPONENTS

Software components are intended to enable reuse of pre-existing components, and thereby reduce production cost and time-to-market [19]. In a component-based approach to software development, instead of building monolithic systems from scratch each time, pre-existing components from a repository are (re)used to build many different systems (Figure 1).

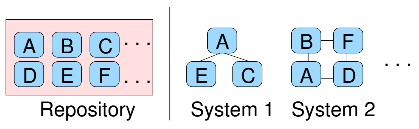


Figure 1: Component-based software development.

A generally accepted view of a software component is a unit of composition [19]. In current component-based approaches [14, 17], components are either objects (as in object-oriented programming) or architectural units (as in software architectures [4]). Exemplars of these approaches are Enterprise JavaBeans [7] and Architecture Description Languages (ADLs) [18] respectively. These kinds of components lack proper composition operators for systematic composition.

We have proposed an alternative approach [15], where proper composition operators are defined. These operators allow us to build systems by composing components hierarchically. Encapsulation and compositionality are the distinguishing characteristics of our approach. They lead to a hierarchical way of building systems, in which a composite is *self-similar* to its sub-components. Self-similarity means that each composite component has the same structure as its sub-components. Self-similarity provides a compositional approach to system construction, and this is an advance over hierarchical approaches such as ADLs which are not compositional.

For verified software, the hierarchical nature of systems means that verification conditions are hierarchical. As a result, in proving a composite of sub-components, the proofs of sub-components can be reused in the proof of the composite. This should enable our approach to scale up.

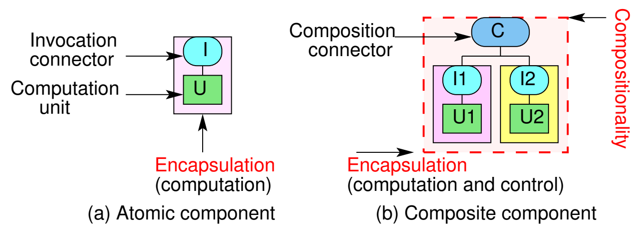


Figure 2: Components and composition in our approach.

2.1 Our Component-based Approach

In our approach (Figure 2), components are constructed from two kinds of basic entities: (i) *computation units*, and (ii) *connectors*. A computation unit U encapsulates computation. It provides a set of methods (or services). *Encapsulation* means that U's methods do not call methods in other computation units; rather, when invoked, all their computation occurs in U. Thus U could be thought of as a class that does not call methods in other classes.

There are two kinds of connectors: (i) *invocation*, and (ii) *composition*. An invocation connector is connected to a computation unit U so as to provide access to the methods of U.

A composition connector encapsulates *control*. It is used to define and coordinate the control for a set of components. For sequencing, we use the *pipe* and *sequencer* connectors, and for branching, we use the *selector* connector.[1] A *pipe* connector that composes components $C_1, \ldots, C_n$ can call methods in $C_1, \ldots, C_n$ in that order, and pass the results of calls to methods in C_i to those in C_{i+1}. A *sequencer* connector is the same as a pipe but does not pass the results of C_i to C_{i+1}. A *selector* connector that composes components $C_1, \ldots, C_n$ simply selects one component depending on a selection condition.

Components are defined in terms of computation units and connectors. There are two kinds of components: (i) *atomic*, and (ii) *composite* (Figure 2). An atomic component consists of a computation unit with an invocation connector that provides an interface to the component. A composite component consists of a set of components (atomic or composite) composed by a composition connector. The composition connector provides an interface to the composite.

Invocation and composition connectors form a hierarchy [16]. This means that composition is done in a hierarchical manner. Furthermore, each composition *preserves* encapsulation. This kind of compositionality is the distinguishing feature of our approach. An atomic component encapsulates *computation* (Figure 2(a)), namely the computation encapsulated by its computation unit. A composite component encapsulates *computation* and *control* (Figure 2(b)). The computation it encapsulates is that encapsulated in its sub-components; the control it encapsulates is that encapsulated by its composition connector. In a composite, the encapsulation in the sub-components is preserved. Indeed, the hierarchical nature of the connectors means that composite components are *self-similar* to their sub-components, as can be clearly seen in Figure 2(b).

In general, a system constructed using our approach consists of a hierarchy of composition connectors sitting atop a flat layer of decoupled atomic components (Figure 3). The hierarchy of composition connectors totally encapsulates the control in the system, whilst the atomic components encapsulate the computation performed by the system. The system thus looks like that in Figure 3, where self-similarity is clearly evident.

[1]We can use a loop connector for looping, but this is a unary connector and the loop must be finite.

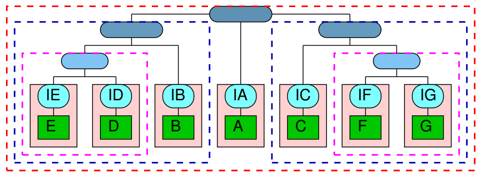


Figure 3: Self-similarity in a generic system.

2.2 Implementation in SPARK

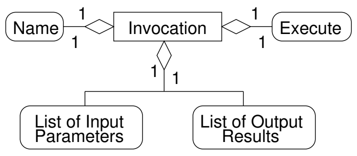
To implement components according to our approach, we use special SPARK packages. Because components in our approach do not call each other, packages for components can only import commonly used packages such as common types, but not other component packages.[2]

A computation unit is implemented as a package, and an invocation connector as a package that imports this package. A composition connector is implemented as a package that imports the component packages that it connects.

An atomic component is implemented as two packages, one for the computation unit and one for the invocation connector. A composite component is implemented as a set of packages that include a package for its composition connector and packages for its sub-components.

The generic design of an invocation connector is shown in Figure 4 (a). Each invocation connector (with a unique 'Name') connects a single computation unit defining a number of methods. Each method has required input parameters ('List of Input Parameters') and provided output results ('List of Output Results'). An invocation connector can call any method defined in the computation unit through an 'Execute' method by providing required input parameters and collecting provided output results. This design can be implemented as an Abstract Data Type (ADT) in SPARK as shown in Figure 4 (b).

A composition connector, be it Pipe, Sequencer or Selector, takes at least two components and returns a composite component whose top-level connector (interface) is this composition connector. The generic design of a Pipe connector is shown in Figure 5 (a). Each Pipe connector connects at least two components in a pre-defined order. Each component has its own top-level connector (interface),



(a)

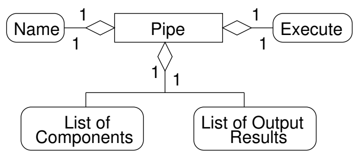
```
type CUMethod is (MethodA, MethodB, . . . );
type CUParam is
     record
          ParamA : ParamTypeA;
          ParamB : ParamTypeB;
          . . .
     end record;
type CUResult is
     record
          ResultA : ResultTypeA;
          ParamB : ResultTypeB;
          . . .
     end record;

type CUInvocationType is
     record
          InvMethod : CUMethod;
          InvParam : CUParam;
          InvResult : CUResult;
     end record;
```

(b)

Figure 4: Invocation connector: design and implementation.

[2]Importing is expressed by the 'with' clause in SPARK.



(a)

```
type CParam is
     record
        ParamA : ParamTypeA;
        ParamB : ParamTypeB;
        . . .
     end record;
type CResult is
     record
        ResultA : ResultTypeA;
        ResultB : ResultTypeB;
        . . .
     end record;

type CPipeType is
     record
        PParam : CParam;
        PResult : CResult;
     end record;
```

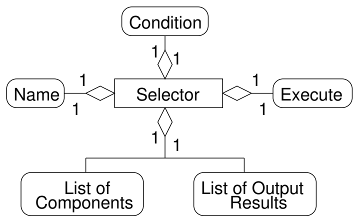
(b)

Figure 5: Pipe connector: design and implementation.

which is either an invocation connector or a composition connector. All the top-level connectors of constituent components form a ‘List of Components’ and their output results form a ‘List of Output Results’. A Pipe connector calls all the ‘Execute’ methods of its constituent components in the pre-defined order through an ‘Execute’ method of its own by providing any required input parameters and collecting provided output results to store in its ‘List of Output Results’. This design can be implemented as an ADT in SPARK as shown in Figure 5 (b).

A Sequencer connector can be designed and implemented in a similar manner; so we omit it here.

The generic design of a Selector connector is shown in Figure 6 (a). Each Selector connector connects at least two components with



(a)

```
type CSelCond is (C1, C2, . . . );
type CParam is
     record
        ParamA : ParamTypeA;
        ParamB : ParamTypeB;
        . . .
     end record;
type CResult is
     record
        ResultA : ResultTypeA;
        ResultB : ResultTypeB;
        . . .
     end record;

type CSelectorType is
     record
        SelCond : CSelCond;
        SelParam : CParam;
        SelResult : CResult;
     end record;
```

(b)

Figure 6: Selector connector: design and implementation.

a pre-defined condition. A Selector connector calls the ‘Execute’ methods of one of its constituent components based on the valuation of its pre-defined condition through an ‘Execute’ method of its own by providing any required input parameters and collecting provided output results to store in its ‘List of Output Results’. This design can be implemented as an ADT in Spark Ada as shown in Figure 6 (b).

To make the task of constructing components and composing them easier in practice, we have implemented two languages CSL and CCL. CSL is used for specifying and generating atomic components, and CCL is used for composing components defined in CSL. Both CSL and CCL are implemented in SPARK, but their semantics is that of our component-based approach.

The development process using CSL and CCL is summarised in Fig. 7. To construct an atomic component A with computation

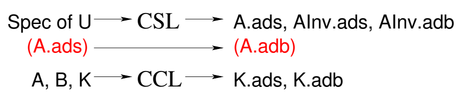


Figure 7: Development process with CSL and CCL.

unit U, the specification of U (pre/post-conditions of its methods) is fed into CSL. CSL then automatically generates the invocation connector for A (`AInv.ads`, `AInv.adb`) and the specification for A (`A.ads`). The body for A (`A.adb`), i.e. the code for U, has to be supplied by the user.

To construct a composite component from components A, B and composition connector K, these are fed into CCL. CCL then automatically generates the specification of the composite (`K.ads`) and the body of the composite (`K.adb`).

In the next section, we will show how our approach can be used to produce verified component-based software in SPARK.

3. VERIFIED SOFTWARE

In SPARK, the functional correctness of a procedure can be verified by using appropriate tools. For this purpose, the procedure has to be specified by a pre-condition and a post-condition. It should also contain suitable assertions in its body. In particular, loops within the procedure body must have suitable assertions that define the loop invariants.

Consider a package X containing a procedure with a specification $\{$(pre-condition) Q, (post-condition) $R\}$, and a body B. The SPARK Examiner can automatically analyse B and identify all the paths in B, provided B contains all the necessary annotations, in particular loop invariants. For each path in B, the Examiner can also automatically generate its pre-condition U and its post-condition V from the specification $\{Q, R\}$ of X and the annotations in B, and hence a verification condition (VC) for the path. The VC is of the form $H \rightarrow C$, where H is a hypothesis and C the conclusion. H is of course just the pre-condition U of the path, and C is the weakest pre-condition for the post-condition V of B.

To prove the correctness of X, the VCs of all the paths have to be proved. The SPARK Simplifier can be used to prove VCs, but often it cannot do so completely automatically, and the human user has to complete the proofs of the undischarged proof obligations interactively using the Proof Checker. The Proof Checker allows user-guided proofs following manually constructed proof rules added by the user. VCs can also be discharged manually by the Review Team.

3.1 Verified Component-based Software

In a generic system constructed by our component-based approach (Figure 3), the paths are hierarchical. This is because all the control in the system is defined by the hierarchy of composition connectors sitting on top of the atomic components. Therefore, all the paths are defined in this hierarchy, and are hierarchical as a consequence. Hierarchical paths mean hierarchical VCs.

Figure 8 illustrates this. It shows one path $P1$ in the system in

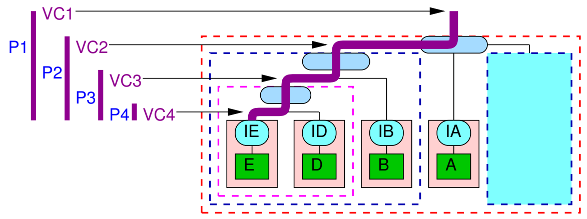


Figure 8: Hierarchical paths and VCs.

Figure 3 (simplified here). $P1$ contains sub-path $P2$, which contains sub-path $P3$, which in turn contains sub-path $P4$. Consequently, the verification condition *VC1* for $P1$ is defined in terms of the verification condition *VC2* for $P2$, which is defined in terms of the verification condition *VC3* for $P3$, which in turn is defined in terms of the verification condition *VC4* for $P4$.

Hierarchical VCs mean that the proof of a VC can reuse the proofs of its sub-VCs. So in Figure 8, once the proof of *VC4* has been done, it can be reused in the proof of *VC3*, which can be reused in the proof of *VC2*, which in turn can be reused in the proof of *VC1*. In general, this means that when constructing a system using our approach, we only have to fully prove the VCs at the lowest level, i.e. at the level of atomic components. Subsequently, at each level of composition, the new VCs can be proved by reusing proofs of VCs at the previous level. This reduces the proof effort at each level, and this is the reason for our belief that our approach can scale up. In the next section, we show a case study that bears out our belief.

4. A MISSILE GUIDANCE SYSTEM

Using the SPARK implementation of our component-based approach, we have experimented on an industrial strength case study, a Missile Guidance system [10], which we obtained from Praxis High Integrity Systems.

The Missile Guidance system is the main control unit for an endo-atmospheric interceptor missile (Fig. 9). The missile system consists of a main control unit (Nav) and input/output. An I/O handler reads data from different sensors (Barometer, Airspeed, Warhead, Fins) and passes them via a bus to corresponding processing units (If_Barometer, If_Airspeed, If_Warhead, If_Fins). The output of these units is used to guide the missile.

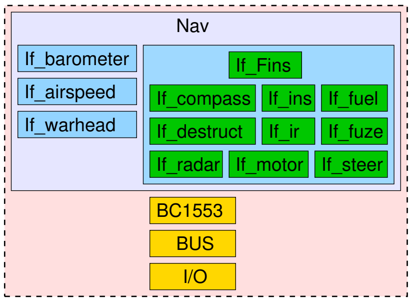


Figure 9: The missile guidance system.

Using our component-based approach, we have implemented a component-based version of the Missile Guidance system. Its architecture is shown in Figure 10. Seq1, Seq1', Seq2 and Seq4 are composite components whose interfaces are sequencer connectors. Sel2 is a composite component whose interface is is a selector connector. Seq3 is a sequencer connector, Sel1 a selector, Pipe1 and Pipe2 are pipe connectors and Loop is a unary iterator.

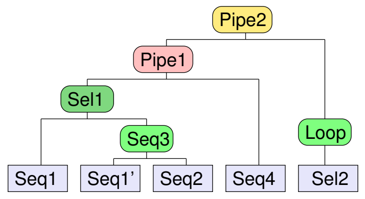


Figure 10: A component-based missile guidance system.

```
package Barometer is
   type BarometerType is
         record
            Height : Integer;
            DZ : Integer;
         end record;
         . . .
   procedure Set_Altitude_Profile(Height : in Integer; DZ : in Integer;
         . . .                              ABarometer : out BarometerType);
   --# pre Height /= 0 and DZ /= 0;
   --# post ABarometer.Height = Height and ABarometer.DZ = DZ;
   . . .
end Barometer;
```

Figure 11: Computation unit BM.

```
package BarometerInvocation is
   type BarometerMethod is (Set_Altitude_Profile, Read_Altitude, . . .);
   type BarometerParam is
         record
            Height : Integer;
            DZ : Integer;
         end record;
   type BarometerResult is
         record
            ABarometer : Barometer;
            Height : Integer;
         end record;
   type BarometerInvType is
         record
            AMethod : BarometerMethod;
            AParam : BarometerParam;
            AResult : BarometerResult;
         end record;
   procedure Execute(AInv : in out BarometerInvType);
   . . .
   --# pre (AInv.AMethod = Set_Altitude_Profile) -->
   . . .     (AInv.AParam.Height /= 0 and AInv.AParam.DZ /= 0);
   --# post (AInv.AMethod = Set_Altitude_Profile) -->
             (AInv.AResult.ABarometer.Height = AInv.AParam.Height
             and AInv.AResult.ABarometer.DZ = AInv.AParam.DZ);
   . . .
end BarometerInvocation;
```

Figure 12: Invocation connector for BM.

In our implementation, the Missile Guidance system contains 246 packages including tools and a test harness, and a total of 30,102 lines of SPARK code including comments and annotations. We reused some existing code from Hilton's implementation, mainly for the construction of computation units, which include airspeed, barometer, steer, etc. These computation units are composed by composition connectors at different levels into composites that are stored alongside atomic components in a repository.

An example of a computation unit is the Barometer ('bm' in Figure 16) in the Seq4 composite component, outlined in Figure 11.

An example of an invocation connector is for the Barometer ('ibm' in Figure 16), outlined in Figure 12.

An example of a sequencer composition connector is that in the

```
package Seq4 is
   type Seq4Param is
      record
         BarInv : BarometerInvocation.BarometerInvType;
         . . .
      end record;
   type Seq4Result is
      record
         BarInvResult : BarometerInvocation.BarometerResult;
         . . .
      end record;
   type Seq4Type is
      record
          SeqParam : Seq4Param;
          SeqResult : Seq4Result;
      end record;
   procedure Execute(APipe : in out Seq4Type);
   . . .
   --# pre (APipe.SeqParam.BarInv.AMethod =
         BarometerInvocation.Set_Altitude_Profile) -->
         (APipe.SeqParam.BarInv.AParam.Height /= 0 and
         and APipe.SeqParam.BarInv.AParam.DZ /= 0);
         . . .
   --# post (APipe.SeqParam.BarInv.AMethod =
       BarometerInvocation.Set_Altitude_Profile) -->
       (APipe.SeqResult.BarInvResult.ABarometer.Height =
       APipe.SeqParam.BarInv.AParam.Height and
       APipe.SeqResult.BarInvResult.ABarometer.DZ =
       APipe.SeqParam.BarInv.AParam.DZ);
       . . .
end Seq4;
```

Figure 13: Sequencer connector Seq4.

Seq4 composite component, outlined in Figure 13.

The VCs of the missile guidance system produced by the SPARK Examiner are indeed hierarchical, as exemplified by the VC for the Seq4 subsystem in Figure 14. This enables us to reuse proofs of sub-VCs when proving a VC. In Figure 14, in the proof of the VCs, (1) is re-written or 'discharged' by (1'), and (1') re-written by (1"), and so on.

This reduces the total proof effort considerably, with proof at each level of composition consisting mainly of reused proofs of at the previous level of composition, as in Figure 8.

We have proved the system completely, using the SPARK Examiner, Simplifier and Checker. The proof obligation summary is shown in Fig. 15. The summary is generated automatically by the

```
Total VCs by type:
                          ------------Proved By---------------
                   Total   Examiner  Simp  Checker  Review  False
Undiscgd
Assert or Post:      969       701    193      181       0      0
Precondition Check     5         0      0        0       0      0
Check Statement        0         0      0        0       0      0
Runtime check:      1494         0   1286      109       0      0
Refinement VCs:      350       350      0        0       0      0
Inheritance VCs:       0         0      0        0       0      0

=======================================================
Totals:             2818      1051   1479      290       0      0
% Totals:                      37%    52%      10%      0%     0%
------------------------End of Semantic Analysis Summary------
```

Figure 15: Proof Obligation Summary of missile system.

SPARK Proof Obligation Summariser (POGS). It is a summary for the VCs: their total number, types, and numbers discharged by each proof tool.

In the proofs of composite components, we succeeded in reusing proofs of sub-components, by virtue of the hierarchical nature of the VCs. We define proof reuse rate for a (composite or atomic) component simply as the ratio of the number of new VCs for the (composition or invocation) connector to the number of VCs in the sub-components (or computation unit). Of course the actual proof effort for each VC is variable, but we believe the ratio of VC numbers does give a first approximation to proof reuse rate.

As an illustration of the proof reuse rates for the component-based missile guidance system, we will show the proof reuse rates for part of the system, viz. the composite component Seq4 in Fig. 10. The subcomponents of Seq4 are shown in Fig. 16, where

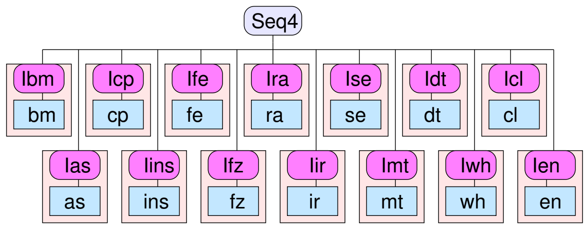


Figure 16: Seq4: part of the missile guidance system.

'Ibm' is the invocation of 'bm' (Barometer), 'Ias' is the invocation of 'as' (Airspeed), etc.

The proof reuse rate for each sub-component of Seq4 is shown in Fig. 17. We can see that the bulk of proof efforts goes into prov-

Package	bm	Ibm	as	Ias	cp	Icp	ins	Iins	fe	Ife	fz	Ifz	ra	Ira
No. of VCs	11	17	11	19	19	31	15	23	13	20	13	20	28	35
Reuse rate	65%		58%		61%		65%		65%		65%		80%	
Package	ir	Iir	se	Ise	mt	Imt	dt	Idt	wh	Iwh	cl	Icl	en	Ien
No. of VCs	28	34	21	28	19	25	12	18	12	18	12	21	30	38
Reuse rate	82%		75%		76%		67%		67%		57%		79%	
Package	Seq4													
No. of VCs	352													
Reuse rate	98%													

Figure 17: Proof reuse rates for Seq4.

ing the computation units of atomic components, but these proofs are only done once and can be reused afterwards. Our component-based approach is able to reuse these proofs effectively, thus reducing the cost of proof efforts of the whole system.

More importantly, this experiment confirms that our component-based approach can scale up, because of proof reuse.

5. CONCLUSION

In this paper we have demonstrated that our component-based approach can be used to construct verified component-based software. In systems constructed using our approach, VCs are hierarchical, in terms of levels of composition. As a result, proof reuse of sub-VCs is possible, thus considerably reducing the overall proof effort. Our approach therefore has the potential to scale up.

We have shown supporting experimental results on a missile guidance system with about 30,000 lines of code. Our implementation is in SPARK, and our experimental results are obtained by using the SPARK proof tools. We have proved the system completely.

In principle, our component-based approach could be implemented in any verification technology, such as JML/Java [13] and Spec#/C# [3], for object-oriented languages. For this experiment we chose SPARK because it is a simpler language and its proof tools have been tried and tested in industrial projects.

Finally, our approach has some similarities with HOOD. In the HOOD Avionics Architecture Description Language (AADL) [8], components are behavioural code of a module, and connections are procedural code of an operation; connections represent functional

```
VC of Seq4 :
    H1 : fld_amethod(fld_barinv(fld_seqparam(apipe))) = set_altitude_profile --> fld_height(fld_aparam(fld_barinv(fld_seqparam(apipe)))) <> 0 .
                                                                                 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                                                                                                       (1)
    H2 : fld_amethod(fld_barinv(fld_seqparam(apipe))) = set_altitude_profile --> fld_dz(fld_aparam(fld_barinv(fld_seqparam(apipe)))) <> 0 .
                                                                                 - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                                                                                                       (2)
    . . .
    Hi : . . .
            -->
    C1 : fld_amethod(fld_barinv(fld_seqparam(apipe))) = set_altitude_profile -->
         fld_height(fld_abarometer(fld_barinvresult(fld_seqresult(apipe)))) =   fld_height(fld_aparam(fld_barinv(fld_seqparam(apipe)))) .
         - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -    - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                                   (3)                                                               (4)
    C2 : fld_amethod(fld_barinv(fld_seqparam(apipe))) = set_altitude_profile -->
         fld_dz(fld_abarometer(fld_barinvresult(fld_seqresult(apipe)))) =   fld_dz(fld_aparam(fld_barinv(fld_seqparam(apipe)))) .
         - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -    - - - - - - - - - - - - - - - - - - - - - - - - - -
                                   (5)                                                           (6)
 VC of barometer's invocation connector (IBM):
     H1 : fld_height(fld_aparam(ainv)) <> 0 .
          - - - - - - - - - - - - - - -
                    (1')
     H2 : fld_dz(fld_aparam(ainv)) <> 0 .
          - - - - - - - - - - - -
                    (2')
     . . .
     Hi : . . .
            -->
     C1: fld_amthod(ainv) = set_altitude_profile -->
         fld_height(fld_abarometer(fld_aresult(ainv))) =   fld_height(fld_aparam(ainv)) .
         - - - - - - - - - - - - - - - - - - - - - - -     - - - - - - - - - - - - - -
                          (3')                                      (4')
     C2: fld_amthod(ainv) = set_altitude_profile -->
         fld_dz(fld_abarometer(fld_aresult(ainv))) =   fld_dz(fld_aparam(ainv)) .
         - - - - - - - - - - - - - - - - - - - - -     - - - - - - - - - - - -
                          (5')                                (6')
VC of barometer's computation unit (BM) :
    H1 : height <> 0 .
         - - - -
          (1'')
    H2 : dz <> 0 .
         - -
        (2'')
    . . .
    Hj : . . .
           -->
    C1: fld_height(upf_dz(upf_height(abarometer, height), dz) = height .
        - - - - - - - - - - - - - - - - - - - - - - - - - - - -   - - - -
                                (3'')                              (4'')
    C2: fld_dz(upf_dz(upf_height(abarometer, height), dz)) = dz .
        - - - - - - - - - - - - - - - - - - - - - - - - - -   - -
                                (5'')                        (6'')
    . . .
```

Figure 14: Hierarchical VCs of component-based missile guidance system.

and structural use relationships of components. This makes HOOD AADL connection a good representation of our composition connector. It would be interesting to investigate the construction of Avionics systems using a HOOD AADL implementation of our approach. In particular, we believe that this would make the verification of Avionics systems easier through proof reuse.

Acknowledgements

We would like to thank the reviewers for helpful and constructive comments that have enabled us to improve the paper considerably.

6. REFERENCES

[1] P. Amey. Correctness by construction: Better can also be cheaper. *CrossTalk (The Journal of Defense Software Engineering)*, pages 24–28, March 2002.

[2] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.

[3] M. Barnett, K. Leino, and W. Schulte. The Spec# programming system: An overview. In *Proc. Int. Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, LNCS 3362*, pages 49–69. Springer, 2004.

[4] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, second edition, 2003.

[5] E. Colbert and B. Lewis. Architecture-centered development of time critical systems with AADL, UML and Ada. In *Proceedings of ACM SIGAda 2003*, 2003.

[6] M. Croxford and R. Chapman. Correctness by construction: A manifesto for high-integrity software. *CrossTalk, The Journal of Defense Software Engineering*, pages 5–8, December 2005.

[7] L. DeMichiel and M. Keith. *Enterprise JavaBeans, Version 3.0*. Sun Microsystems, 2006.

[8] P. Dissaux. Using the AADL for mission critical software development. In *Proceedings of ERTS 2004*, `http://la.sei.cmu.edu/aadlinfosite/AADLPublications&Presentations.html`, 2004.

[9] A. Hall and R. Chapman. Correctness by construction: Developing a commercial secure system. *IEEE Software*, pages 18–25, Jan/Feb 2002.

[10] A. Hilton. *High Integrity Hardware-Software Codesign*. PhD thesis, The Open University, April 2004.

[11] T. Hoare and J. Misra. Verified software: theories, tools, experiments — vision of a grand challenge project. In *Proceedings of IFIP working conference on Verified Software: theories, tools, experiments*, 2005.

[12] IFIP TC2 working conference on Verified Software: Theories, Tools, Experiments, 10-13 October 2005, ETH Zürich, Switzerland. `http://vstte.ethz.ch/`.

[13] The Java Modeling Language (JML) Home Page. `http://www.cs.iastate.edu/$\sim$leavens/JML.html`.

[14] K.-K. Lau. Software component models. In *Proc. 28th Int. Conf. on Software Engineering*, pages 1081–1082. ACM Press, 2006.

[15] K.-K. Lau, M. Ornaghi, and Z. Wang. A software component model and its preliminary formalisation. In F. de Boer *et al.*, editor, *Proc. 4th Int. Symp. on Formal Methods for Components and Objects, LNCS 4111*, pages 1–21. Springer-Verlag, 2006.

[16] K.-K. Lau, P. Velasco Elizondo, and Z. Wang. Exogenous connectors for software components. In G. Heineman *et al.*, editor, *Proc. 8th Int. Symp. on Component-based Software Engineering, LNCS 3489*, pages 90–106. Springer, 2005.

[17] K.-K. Lau and Z. Wang. A survey of software component models. Second edition, Pre-print CSPP-38, School of Computer Science, The University of Manchester, May 2006. `http://www.cs.man.ac.uk/cspreprints/PrePrints/cspp38.pdf`.

[18] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.

[19] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, second edition, 2002.