# Component-based Construction of Concurrent Systems with Active Components

Kung-Kiu Lau and Ioannis Ntalamagkas
School of Computer Science, The University of Manchester
Manchester M13 9PL, UK
{kung-kiu,i.ntalamagkas}@cs.manchester.ac.uk

***Abstract*—We have defined a software component model which specifies components and composition operators. These operators coordinate and encapsulate concurrency between components. In this paper we explain how our model can be used to specify and construct concurrent systems in a hierarchical manner that is amenable to compositional reasoning. In particular, we extend previous work on active components via introducing new, concurrent connectors and show how they can be used together for compositionally constructing concurrent systems.**

***Keywords*-active components; concurrent systems; hierarchical composition; compositionality;**

## I. Introduction

Component-based software development [21] aims to build systems from pre-existing components (possibly from a repository). Instead of developing monolithic systems completely from scratch, the emphasis is on re-using software units that have already been built by independent component developers, by assembling them into systems. The cornerstone of a component-based methodology is the underlying component model [8], [14]. A component model defines the syntax and semantics of components, and mechanisms or operators for their composition. Generally speaking, components represent computation or behaviour, and their composition represents their interaction. Ideally, composition should be defined algebraically, so that (algebraic) compositionality would follow suit. Also, explicitly defined composition operators as entities in their own rights would be desirable, because they would provide a 'toolkit' for assembling components into systems.

We have introduced a component model [13], [11] that is different from current models, in particular ADLs [18], [3], in that in our model whilst we do separate components from their interaction, we also have explicit composition operators that define component interaction. In this paper, we extend our model so that it can be used to define and construct concurrent systems.

## II. Our component model

We have proposed a component model [13], [11] in which a single sequential computation process can be defined (and implemented). In [10], active components were defined. In this paper we extend our model and introduce concurrent connectors, i.e. connectors that can handle and/or define concurrent processes. In the rest of the paper, when we refer to our component model, we mean our extended component model with concurrency.

In our model there are three basic entities, (i) *computation units*, (ii) *active computation processes* and (iii) *connectors*. They are used to construct two kinds of components: (i) *atomic*, and (ii) *composite* (Fig. 1). Computation units are

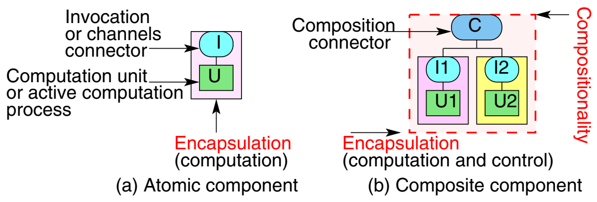

Invocation or channels connector
I
U
Computation unit or active computation process
Encapsulation (computation)
(a) Atomic component
Composition connector
C
I1
I2
U1
U2
Compositionality
Encapsulation (computation and control)
(b) Composite component

Figure 1. Our component model.

passive entities that when invoked perform some computation and then terminate and return the result of the computation to the invoker. Active computation processes on the other hand execute independently and typically never terminate while constantly interacting with their environment. *Invocation* and *channels* connectors in our model provide access to computations units and active computation processes respectively, thus defining passive and active atomic components (Section II-A). *Composition* connectors compose atomic components into *composite components* Section II-B). Composite components can be further composed with other (atomic or composite) components, until the whole system is built.

A distinguishing characteristic of our component model is that computation units and active computation processes encapsulate *computation*, whilst composition connectors encapsulate *control*. Furthermore, composition preserves encapsulation. All this is shown in Figure 1.

### A. Atomic components

Figure 2 shows the structure of a passive atomic component and a typical concurrent execution. A passive atomic component does not define its own executing process. It

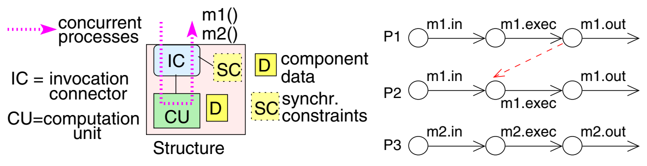


Figure 2. Passive atomic component.

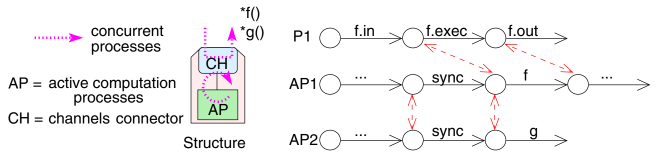


Figure 3. Active atomic component.

consists of a (passive) *computation unit* and an *invocation connector*. The computation unit contains a set of methods or operations, `m1()` and `m2()` in the figure, and performs them when invoked by the invocation connector. The invocation connector provides the interface for the component and is activated by processes from outside the component.These processes are initiated by external users of the system. We have extended the invocation connector so that the external processes can be executed in parallel, as is explained below.

In Figure 2, three processes $P_1$, $P_2$ and $P_3$ access the atomic component concurrently, the first two invoking `m1` and the third one invoking `m2`. As shown in the figure, for each method we consider that the only actions of interest are the input (*m.in*), the execution (*m.exec*) and the output (*m.out*) of the method, which correspond to actions of the concurrent processes. Although these processes are independent, they may need to synchronise before actually executing a method (because components may contain their own data [12], as also shown in the figure). The dotted arrow in the figure denotes synchronisation. For example, $P_1$ and $P_2$ synchronise before executing method `m1`, and $P_2$ may not execute `m1` unless $P_1$ executes `m1` first. For describing the synchronisation requirements for the computation unit we introduce into our model the constructs available in ESP (Extending Scheduling Predicates) [17]. ESP allows for fine-grained concurrency control, and usual synchronisation mechanisms like semaphores and monitors are too low-level when compared to ESP. Finally, ESP can be readily adapted to the context of our component model.

The synchronisation requirements are written using a declarative language in a *synchronisation constraints* file that is read and enforced by the invocation connector. Each method of the computation unit is associated with a predicate that when true allows the process calling the method to execute it. Counters are used to define these predicates. For each method of the computation unit, these counters count the occurrence of each of the actions that comprise a method call (*m.in*, *m.exec*, *m.out*). For example, in Figure 2, the synchronisation constraint for method `m1` is `EXEC m1 EQ 0`, which means that in order for a process to execute `m1`, no other process may be executing it. For `m2` the synchronisation constraint is trivially `TRUE`.

An active atomic component consists of *active computation processes* and a *channels connector*. Figure 3 shows the structure of an active atomic component and a typical execution. The active computation processes ($AP_1$ and $AP_2$ in the figure) perform computation continuously. External processes ($P_1$ in the figure) interact with $AP_1$ and $AP_2$ via the channels connector. The exact interaction mechanism has been described in [10]. In that work the active atomic component could interact with a single external process. We present below a brief overview of active atomic components and we extend their definition so that they may serve concurrent external processes.

We define active computation processes as a non-empty set of CSP [9] processes executing in parallel, as for example processes $AP_1$ and $AP_2$. Informally, every CSP process consists of a set of synchronous channels used to communicate with other processes, for example processes $AP_1$ and $AP_2$ synchronise on channel *sync*. During this synchronisation, processes may exchange data. The active computation processes inside the active component may also execute their actions in parallel, i.e. without any synchronisation. The channels connector provides the interface for the component and access to some channels of the CSP processes. External processes may interact with some of the channels of the composed CSP processes by invoking the active component with the name of the CSP channel they want to interact with, along with any input parameters. The channels connector then forwards these calls to the CSP processes, and the external, calling processes synchronise with them. For the active atomic component in Figure 3, we consider that the only channels that external clients may interact with are channels $f$ and $g$ of the processes $AP_1$ and $AP_2$ respectively. These channels are exposed as methods `*f()` and `*g()` respectively, in the interface of the atomic component, where the asterisk denotes that these methods belong to an active component. The channels connector may be executed by concurrent external processes that wish to interact with the active computation processes. In the figure we only show a single external process $P_1$ calling method `f`. When $P_1$ calls method `f`, its execution ($f.exec$) will synchronise with the execution of channel $f$ of the CSP process $AP_1$.

## B. Composite components

In our model we define *composition operators* or *composition connectors* that compose passive and active atomic components into composite components. This is shown in Figure 1. We use explicit composition operators to compose atomic components via their interfaces into composite components. Our composition operators in [13], [11] have been enhanced for allowing composite components to be accessed by parallel processes.Additionally, we define a new composition operator that explicitly creates concurrent processes.

A distinguishing feature of our composition operators is that they preserve encapsulation in the sub-components. This means that the connected sub-components are only accessed via their interfaces and that no other connector may access these sub-components directly. In addition to that, all our components are *similar*, in the sense that they define a uniform interface through which they may be invoked, i.e. a set of methods. These facts make hierarchical composition possible: a hierarchy of composite components is created incrementally, until the whole system is built, see Figure 1(b).

Composition operators as connectors that coordinate *control* (and *data*) flow between the sub-components [13], [11]. A composition connector *encapsulates* control. It is used to define and coordinate the control for a set of components. For example, for sequencing we use the *pipe* and *sequencer* connectors, and for branching, we use the *selector* connector. 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 $C_k$, $k > i$. A *sequencer* connector is the same as a pipe but does not pass the results of $C_i$ to $C_k$. A *selector* connector simply selects one component depending on a selection condition.

We introduce the *non-deterministic* selector, which is a selector that non- deterministically chooses to execute one of the lower level components that it is connected to. It is similar to the `select` statement of Ada [22] and to the non-deterministic `ALT` of Occam (when at least two of the guards evaluate to true) [6]. It models the situation where the actual branch followed does not affect the correctness of the program.As with the usual meaning of non-determinism, it does not mean that a random choice is made, it only means that the system developer is allowed to choose any reasonable algorithm to make the selection, and users of the composite must not rely on the algorithm to determine the correctness of their code.

For the above connectors, each of the process accessing the composite component is served sequentially. We introduce the *cobegin* composition connector as shown in Figure 4. The cobegin connector composes $n > 1$ components.

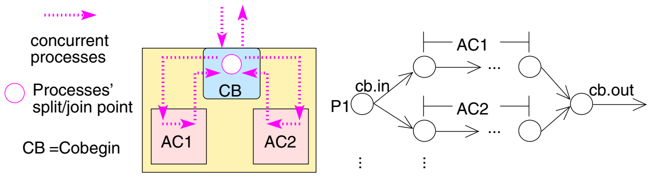


Figure 4. The cobegin composition connector.

For each incoming process to the composite component, it branches into $n$ processes, each one executing in parallel one of the sub-components. Before the cobegin returns, the executing processes must join as shown in the figure, and the union of the results is returned to the calling connector. As also evident in the figure, concurrently executing components do not communicate with each other because that would break their encapsulation. Using the cobegin connector, the system developer can explicitly create new processes and then merge them in the same connector. Allowing process creation and merging to occur in separate connectors would make our approach not compositional. For example, for a single process starting the execution of the component defined using a process-creating connector, more than one processes would return.

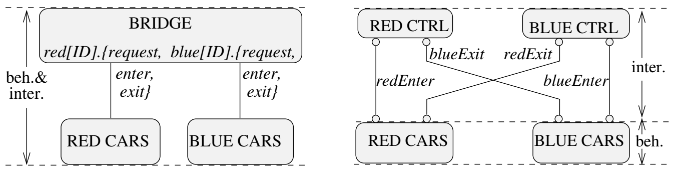
## III. Compositionality

An important property of any component-based approach is compositionality. Indeed it is this property that enables our model to be used for practical, hierarchical construction of component-based systems. Apart from system construction, compositionality is also important for properties related to system construction based on component composition. We have already explained that our model is compositional with respect to encapsulation (of computation and control). In this section, we demonstrate the compositionality of our model with respect to reasoning.

For a constructed system, compositional reasoning allows the system developer to infer overall system properties from its components properties. It is clearly a great asset for reducing the complexity of concurrency verification. A fundamental requirement for compositional reasoning is that the observable behaviour of each subcomponent is completely independent from the observable behaviour of the other subcomponents [5]. This is an intrinsic property of our model.Therefore, using our model for system construction makes it possible to apply compositional reasoning rules which allow to infer the composite's specification based on the specification of the subcomponents. Additionally, the complete separation of computation from interaction in our model allows us to adjust the specification of the composite component by simply changing the interaction pattern used for composition and not the composed subcomponents. The following example illustrates these points.

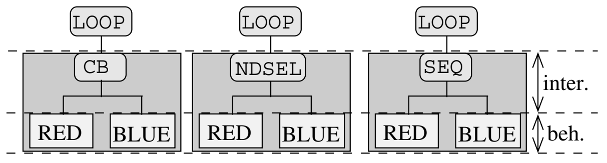
### A. An Example

To illustrate the compositionality aspect of our model and the importance of separating component behaviour from component interaction, we present an alternative way to construct the *Single-Lane Bridge Problem*, which was originally presented in [15] and revisited in [23]. The problem states that there is a narrow bridge that on its two sides there are blue and red cars waiting to cross. Because the bridge is narrow, cars can only pass in one direction concurrently and no overtaking is allowed. In [15], blue and red cars are



(a) Original approach [15] (b) Approach presented in [23]

Figure 5. Single-lane cars-bridge problem.



(a)Cobegin (b) ND−Selector (c) Sequencer

Figure 6. Cars-bridge in our model.

defined using FSP, and so is the bridge controller (Fig.5(a)). The specification is model checked and when errors are found, via successive corrections of the model (i.e. redefinition of both cars and the bridge) the final specification is produced. The fact that both the cars and the bridge need to be redefined in order to get the correct solution is not surprising, because in FSP behaviour is not separated from interaction. Therefore for the interactions *request*, *enter* and *exit* the bridge both components need to synchronise. As a result, the way blue and red cars interact does not only depend on the bridge controller, it also depends on the cars themselves. As we show later, the way blue and red cars are defined should not affect their interaction.

In [23], the authors define a rich set of synchronisation connectors (effectively different channel and port types) that enforce variations of synchronous/ asynchronous communication between concurrently executing components. In this work, channel ports semantically belong to connectors, and components may only exchange messages with attached connector ports. In the proposed solution, components are the blue cars, the red cars, a blue controller and a red controller (Fig.5(b)). When these components are connected with appropriate connectors, they get the correct behaviour of their system. In this work, there is a clear separation between component behaviour and interaction. However, as can be seen in the figure, there is not a consistent treatment of components. Components can be used either to specify behaviour, or to specify interactions between other components. This inconsistency means that some of the components are ad hoc entities created to specify interactions for a specific system. This contrasts the view in CBSE where components are predefined entities residing in a repository. Additionally, their approach is not compositional, i.e. their is not a composition operator for composing components into composite ones.

In our approach, we build two active components representing the blue and red cars respectively, which are derived from the same CSP specification.The controller that decides when the cars pass the bridge is built using our connectors. The high-level description of our system is shown in Fig.6. There are two active components, representing the red and blue cars waiting to cross the bridge. Different composition connectors give different overall behaviour for the composed system. For example, when the *cobegin* is used (Fig.6(a)) then both blue and red cars enter the bridge at the same time, thus leading to collision. This solution is therefore unsafe. When a *non-deterministic selector* is used (Fig.6(b)), cars of a certain colour may dominate the bridge. This solution is therefore unfair. Finally, the correct solution is shown in Fig.6(c) where the sequencer forces blue and red cars to alternate on the bridge. This solution is both safe and fair.

## IV. Related work

Related work consists of component models and ADLs that can be used for modelling and building concurrent systems and that use a formalism for specifying and verifying temporal properties. As already discussed, most existing component models and ADLs offer no formal mechanisms for specifying and checking temporal properties of concurrent systems.

The work presented in [23], formalises a rich set of synchronisation connectors for data flow. However, as shown in Fig.5(b) control is exercised by components as well as connectors. Therefore slight changes in the system requirements, may result in these changes propagating to components. In our approach, since control is only exercised by connectors, changes in the system requirements remain localised in the connectors, and it is therefore only necessary to reconfigure connector composition.

The ADLs Wright [1] and Darwin [16], are formalised in CSP and FSP/$\pi$-calculus respectively. Wright treats connectors as first class entities that connect components based on port-to-port connections. Wright connectors are not predefined, and every connector has a part named *Glue code* that describes how data on the input port(s) are delivered to the output port(s), therefore hindering the plug-and-play development of component based systems. Darwin ADL on the other hand, does not treat connectors as first class entities, resulting in a tight coupling among components. Components follow the usual interaction mechanism through ports connected via synchronous channels. Additionally, in these approaches the whole system is specified in the same formalism, and components of different nature cannot be combined in the same system.

The merits of applying different formalisms to different parts of a system have been recognised in [20], [4]. In this work, the authors apply CSP processes to coordinate and control B-Machines; B-Machines do all computation and deal with system's data. However, the only controllers are CSP processes, which are too low-level for constructing large software systems. Additionally, there is not a compositional approach for building larger systems out of smaller ones and their approach to building controllers is *ad hoc*.

There has been early work [7] in recognising the benefits from separating coordination and computation as separate concerns in software, which led to the development of coordination languages [19]. Reo ([2]) is a coordination language that defines a rich set of low-level connector primitives for coordinating data flow among components. Control lies within components and connectors transfer passive data, i.e. data that do not transfer control. In our approach control starts from connectors and the connectors decide how control (and data) flow in the system. Additionally, Reo offers no advantages in modelling concurrent systems when compared to CSP in terms of expressive power and formal verification.

## V. Conclusions

In this paper we have proposed a model for the component-based construction of concurrent systems. Our model clearly separates the behaviour (or computation) and the interaction parts of a system. We use atomic components to perform computation, and explicit composition operators that define and encapsulate concurrent interactions among components. The composition operators presented define reusable interaction patterns and provide a 'toolkit' for assembling components into systems. These properties clearly distinguish our approach from related works that do not encompass one or more of them. In addition to that, these properties make our work amenable to compositional verification. The inclusion of active CSP components adds to the expressive power of our model, and a wider range of systems can be defined and implemented.

Future work includes formalisation of our connectors and components to capture their executable behaviour. From this work we will gain the ability to reason about the properties of a concurrent system compositionally. Based on the temporal properties of single components and composition connectors, we will be able to deduce the temporal properties of composite components and eventually of the whole system. This is contrasts most related component-based approaches, where model checking the specification of the whole system is the usual approach.

## References

[1] R. J. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Sch. of Comp. Sc., Carnegie Mellon, 1997.

[2] F. Arbab. Abstract Behavior Types: A Foundation Model for Components and Their Composition. *Science of Computer Programming*, 55:3–52, 2005.

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

[4] M. Butler and M. Leuschel. Combining CSP and B for specification and property verification. In *FM*, *LNCS 3582*, pages 221–236. Springer, 2005.

[5] W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers, editors. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2001.

[6] S. Ericsson-Zenith. *Occam 2 Reference Manual*. Prentice-Hall Intern. Series in Comp. Sc.. Prentice Hall, 1987.

[7] D. Gelernter and N. Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, 1992.

[8] G. Heineman and W. Councill, editors. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.

[9] C. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice Hall, 1985.

[10] K.-K. Lau and I. Ntalamagkas. A compositional approach to active and passive components. In *Proc. EUROMICRO-SEAA 2008*, pages 76–83. IEEE, 2008.

[11] K.-K. Lau, M. Ornaghi, and Z. Wang. A Software Component Model and its Preliminary Formalisation. In *Proc. FMCO, LNCS 4111*, pages 1–21. Springer-Verlag, 2005.

[12] K.-K. Lau and F. Taweel. Data encapsulation in software components. In H. Schmidt *et al.*, editor, *Proc. CBSE 2007, LNCS 4608*, pages 1–16. Springer-Verlag, 2007.

[13] K.-K. Lau, P. Velasco Elizondo, and Z. Wang. Exogenous connectors for software components. In *Proc. CBSE 2005, LNCS 3489*, pages 90–106, 2005.

[14] K.-K. Lau and Z. Wang. Software component models. *IEEE Trans. on Software Engineering*, 33(10):709–724, 2007.

[15] J. Magee and J. Kramer. *Concurrency: State Models & Java Programs*. John Wiley & Sons, 1999.

[16] J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour analysis of software architectures. In *WICSA*, *IFIP 140*, pages 35–50. Kluwer, 1999.

[17] C. McHale. *Synchronisation in concurrent, object-oriented languages: expressive power, genericity and inheritance*. PhD thesis, University of Dublin, Trinity College, 1994.

[18] N. Medvidovic and R. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. Software Eng.*, 26(1):70–93, 2000.

[19] G. A. Papadopoulos and F. Arbab. *Coordination Models and Languages*, vol. 46: The Eng. of Large Systems of *Adv. in Computers*, pages 329–400. Academic Press, 1998.

[20] S. Schneider and H. Treharne. Communicating B Machines. In *ZB*, *LNCS 2272*, pages 416–435. Springer, 2002.

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

[22] T.S. Taft and R.A. Duff, editors. *Ada 95 Reference Manual: Language and Standard Libraries, International Standard ISO/IEC 8652:1995(E), LNCS1246*. Springer Verlag, 1997.

[23] S. Wang, G. Avrunin, and L. Clarke. Architectural Building Blocks for Plug-and-Play System Design. In *CBSE*, *LNCS 4063*, pages 98–113. Springer, 2006.