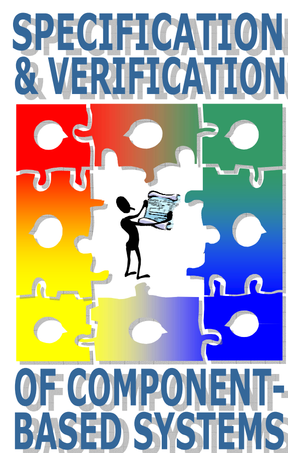
# SAVCBS 2003
# Specification and Verification of Component-Based Systems




*ESEC/FSE 2003*
*9th European Software Engineering Conference*
*and*
*11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*
*Helsinki, Finland*
*September 1-5, 2003*

Technical Report #03-11, Department of Computer Science, Iowa State University

# SAVCBS 2003 PROCEEDINGS

---

## Specification and Verification of Component-Based Systems

## http://www.cs.iastate.edu/ SAVCBS/

September 1-2, 2003
Helsinki, Finland

Workshop at ESEC/FSE 2003
9th European Software Engineering Conference
and
11th ACM SIGSOFT Symposium on the Foundations of Software Engineering

# SAVCBS 2003
# TABLE OF CONTENTS


**ORGANIZING COMMITTEE** vii

**WORKSHOP INTRODUCTION** ix

**INVITED PRESENTATIONS** 1

**Invited Presentation** 2
*Manfred Broy (Technical University Munich)*

**Supporting Model-driven Development of Component-based Embedded Systems with Cadena** 3
*Matthew B. Dwyer (Kansas State University)*

**PAPERS** 5

**SESSION 1**

**Failure-free Coordinator Synthesis for Correct Components Assembly** 6
*Paola Inverardi and Massimo Tivoli (University of L'Aquila)*

**Proof Rules for Automated Compositional Verification through Learning** 14
*Howard Barringer (University of Manchester), Dimitra Giannakopoulou (RIACS/USRA), and Corina S. Păsăreanu (Kestrel Technology LLC)*

**SESSION 2**

**Behavioral Substitutability in Component Frameworks: A Formal Approach** 22
*Sabine Moisan, Annie Ressouche (INRIA Sophia Antipolis), and Jean-Paul Rigault (I3S Laboratory)*

**An Assertion Checking Wrapper Design for Java** 29
*Roy Patrick Tan and Stephen H. Edwards (Virginia Tech)*

**SESSION 3**

**An Approach to Model and Validate Publish/Subscribe Architectures** 35
*Luca Zanolin, Carlo Ghezzi, and Luciano Baresi (Politecnico di Milano)*

**Timed Probabilistic Reasoning on UML Specialization for Fault Tolerant Component Based Architectures** 42
*Jane Jayaputera, Iman Poernomo, and Heinz Schmidt (Monash University)*

**SESSION 4**
**Modelling a Framework for Plugins** 49
*Robert Chatley, Susan Eisenbach, and Jeff Magee (Imperial College London)*

**Form-Based Software Composition** 58
*Markus Lumpe (Iowa State University) and*
*Jean-Guy Schneider (Swinburne University of Technology)*

**SESSION 5**
**Algorithmic Game Semantics and Component-Based Verification** 66
*Samson Abramsky, Dan R. Ghica, Andrzej S. Murawski, and C.-H. Luke Ong (Oxford University)*

**POSTERS** 74
**Bridging the Gap between Acme and UML 2.0 for CBD** 75
*Miguel Goulão and Fernando Brito e Abreu (Faculdade de Ciências e Tecnologia—UNL)*

**Abstract OO Big O** 80
*Joan Krone (Denison University) and W. F. Ogden (The Ohio State University)*

**Ontology-based Description and Reasoning for Component-based Development on the Web** 84
*Claus Pahl (Dublin City University)*

**Modeling Multiple Aspects of Software Components** 88
*Roshanak Roshandel and Nenad Medvidovic*
*(University of Southern California, Los Angeles)*

**Reasoning About Parameterized Components with Dynamic Binding** 92
*Nigamanth Sridhar and Bruce W. Weide (The Ohio State University)*

**DEMONSTRATIONS** 96
**Specifications in the Development Process: An AsmL Demonstration** 97
*Mike Barnett (Microsoft Research)*

**Mae: An Architectural Evolution Environment** 99
*Roshanak Roshandel (University of Southern California, Los Angeles)*

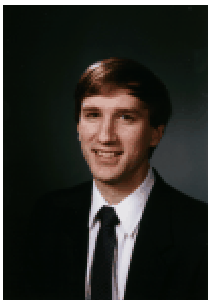
**Runtime Assertion Checking Using JML** 101
*Roy Patrick Tan (Virginia Tech)*

# SAVCBS 2003
# ORGANIZING COMMITTEE

**Mike Barnett (Microsoft Research, USA)**
Mike Barnett is a Research Software Design Engineer in the Foundations of Software Engineering group at Microsoft Research. His research interests include software specification and verification, especially the interplay of static and dynamic verification. He received his Ph.D. in computer science from the University of Texas at Austin in 1992.

**Stephen H. Edwards (Dept. of Computer Science, Virginia Tech, USA)**
Stephen Edwards is an assistant professor in the Department of Computer Science at Virginia Tech. His research interests are in component-based software engineering, automated testing, software reuse, and computer science education. He received his Ph.D. in computer and information science from the Ohio State University in 1995.

**Dimitra Giannakopoulou (RIACS/NASA Ames Research Center, USA)**
Dimitra Giannakopoulou is a RIACS research scientist at the NASA Ames Research Center. Her research focuses on scalable specification and verification techniques for NASA systems. In particular, she is interested in incremental and compositional model checking based on software components and architectures. She received her Ph.D. in 1999 from the Imperial College, University of London.



**Gary T. Leavens (Dept. of Computer Science, Iowa State University, USA)**
Gary T. Leavens is a professor of Computer Science at Iowa State University. His research interests include programming and specification language design and semantics, program verification, and formal methods, with an emphasis on the object-oriented and aspect-oriented paradigms. He received his Ph.D. from MIT in 1989.

**Program Committee:**
Luca de Alfaro (University of California, Santa Cruz)
Mike Barnett (Microsoft Research)
Edmund M. Clarke (Carnegie Mellon University)
Matthew Dwyer (Kansas State University)
Stephen H. Edwards (Virginia Tech)
Dimitra Giannakopoulou (RIACS /NASA Ames Research Center)
Gary T. Leavens (Iowa State University)
K. Rustan M. Leino (Microsoft Research)
Jeff Magee (Imperial College, London)
Heinz Schmidt (Monash University)
Wolfram Schulte (Microsoft Research)
Natalia Sharygina (Carnegie Mellon University)
Murali Sitaraman (Clemson University)
Kurt C. Wallnau (CMU Software Engineering Institute)
Bruce Weide (Ohio State University)

**Sponsors:**

Microsoft® Research

# SAVCBS 2003 WORKSHOP INTRODUCTION

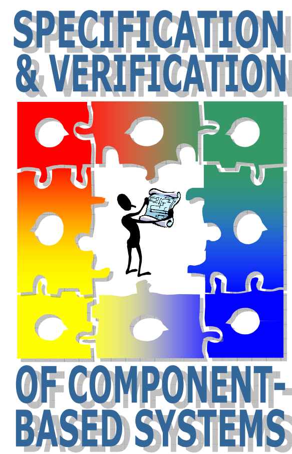
This workshop is concerned with how formal (i.e., mathematical) techniques can be or should be used to establish a suitable foundation for the specification and verification of component-based systems. Component-based systems are a growing concern for the software engineering community. Specification and reasoning techniques are urgently needed to permit composition of systems from components. Component-based specification and verification is also vital for scaling advanced verification techniques such as extended static analysis and model checking to the size of real systems. The workshop will consider formalization of both functional and non-functional behavior, such as performance or reliability.

This workshop brings together researchers and practitioners in the areas of component-based software and formal methods to address the open problems in modular specification and verification of systems composed from components. We are interested in bridging the gap between principles and practice. The intent of bringing participants together at the workshop is to help form a community-oriented understanding of the relevant research problems and help steer formal methods research in a direction that will address the problems of component-based systems. For example, researchers in formal methods have only recently begun to study principles of object-oriented software specification and verification, but do not yet have a good handle on how inheritance can be exploited in specification and verification. Other issues are also important in the practice of component-based systems, such as concurrency, mechanization and scalability, performance (time and space), reusability, and understandability. The aim is to brainstorm about these and related topics to understand both the problems involved and how formal techniques may be useful in solving them.

The goals of the workshop are to produce:

1. An outline of collaborative research topics,
2. A list of areas for further exploration,
3. An initial taxonomy of the different dimensions along which research in the area can be categorized. For instance, static/dynamic verification, modular/whole program analysis, partial/complete specification, soundness/completeness of the analysis, are all continuums along which particular techniques can be placed,
4. A web site that will be maintained after the workshop to act as a central clearinghouse for research in this area, and
5. A special issue of the journal *Formal Aspects of Computing* (published by Springer Verlag). The journal issue will invite revised and expanded versions of selected papers from this and the previous SAVCBS workshop.

# SAVCBS 2003
# INVITED PRESENTATIONS



# Invited Presentation

Manfred Broy
Technical University Munich

# Supporting Model-driven Development of Component-based Embedded Systems with Cadena

Adam Childs, Xianghua Deng, Matthew B. Dwyer, Jesse Greenwald, John Hatcliff, Prashant Kumar, Georg Jung, Venkatesh Ranganath, Robby, Gurdip Singh
Department of CIS
Kansas State University
dwyer@cis.ksu.edu

## ABSTRACT

Developers of modern distributed, real-time embedded systems are increasingly employing component-based middleware frameworks to cope with the ever increasing size and complexity of mission requirements. Frameworks, such as CORBA and its component model (CCM), raise the level of abstraction at which systems are programmed by directly supporting certain essential non-functional requirements of these applications, such as distribution, and through add-on capabilities, such as services that provide event-driven execution. We describe how such frameworks can be used as a foundation for providing even higher-levels of abstraction in system development that can be leveraged throughout the entire software development process. Specifically, we outline the key goals and capabilities of Cadena, an integrated environment that supports the model-driven development of CORBA-based real-time embedded systems.

## 1. MODERN EMBEDDED SOFTWARE

As the processing power available in embedded platforms continues to increase, so too does the demand for increasingly capable software to meet challenging mission requirements. Functionality that once was ”off loaded“ to non-embedded computing nodes is now routinely being deployed in embedded devices. The problems of developing systems that reliably meet stringent timing requirements were hard enough, but now embedded systems developers are confronting the same problems of scale, complexity and distribution that trouble developers in more mainstream application domains. Consequently embedded system developers are adopting standardized component-based development frameworks, such as CORBA, to meet those challenges and adapting them to address timeliness requirements (e.g., [1]). We believe that such frameworks can be enhanced to provide more effective support for the development of highly-reliable embedded software.

To make our discussion more concrete, consider the “push” model of computation used in the BoldStroke middleware [4] which is an adaptation of CORBA. In such a model, a component subscribes to events that signal the availability of data that comprise its inputs, when triggered a component access that data (either from the event payload or via component method calls), it then perform internal calculations and if new values result publishes an event indicating their availability.

For such systems, it is often convenient to divide development into *component development*, where generic and application specific functionality is implemented and packaged with CCM Interface Definition Language (IDL) defined interfaces, and *component integration*, where potentially large numbers of components are instantiated and assembled to fulfill the overall system requirements. This division has the advantage that component development can be reduced, in the best case, to implementation of small simple sequential blocks of code. Unfortunately, component integration remains extremely difficult.

CCM IDL and CCM-compliant middleware, such as OpenCCM [2], provide only a limited form of modeling (i.e., defining the input/output structure of component types), yet they are attractive since they automate significant amounts of platform specific coding. To develop a complete working system, however, additional details must be provided as code, such as, component instantiation, event subscription, and component method synchronization. These tasks fall to the component integrator who requires an understanding of global system behavior to define those details. Unfortunately, despite the component nature of middleware frameworks, global reasoning requires consideration of assemblies of component instances; modular reasoning is not well supported. Perhaps surprisingly in event-driven systems, such as the push model described above, even simple control flow relationships, that are syntactically apparent in many system descriptions, are obfuscated by the inversion of control provided by the middleware. This makes it very difficult to to determine the functional properties of a system much less properties related to real-time, data-coherence, correct synchronization, etc. *Automated analysis of global properties of component assemblies must be available early in the development process for effective component integration.*

It is sometimes necessary to compromise natural component and framework abstractions in order to achieve essential system correctness properties or desired levels of performance. For example, when a system contains some instances of a component type that require synchronization

Specification and Verification of Component-based Systems, Sept. 2003

and others that do not, this non-functional aspect forces developers to define variants of the common functional component interface. Fast-path middleware optimizations can often be performed when knowledge about component assembly (e.g., component co-location) is available. Current approaches that delay optimization to run-time miss opportunities for even greater performance that could be achieved by static exploitation of information about component assemblies that is not available in existing IDL. *Component and system modeling notations must be enriched to include additional semantics to enable effective analysis and to increase the scope and quality of system synthesis.*

In this context, we have developed Cadena an integrated environment intended to support the definition, validation and synthesis of component-based, distributed, real-time, embedded software from high-level structural and behavioral models [3].

## 2. CADENA GOALS AND CAPABILITIES

The primary goal of Cadena is to address the problems encountered during component integration by (*i*) providing developers with feedback on system correctness properties early in the development process, (*ii*) enriching the existing synthesis technologies in middleware frameworks to generate more of the application code-base, and (*iii*) exploiting information about component assemblies to drive automatic performance optimization.

Our strategy is to exploit existing IDL as a basis for layering additional light-weight specification forms. The intent is to provide a means of balancing developer investment with accrued benefit to help address the high entry-barrier of using formal notations that typically stifles their adoption. Specifically, we have developed a suite of specification forms that describe component *instantiation* (i.e., naming and parameterization of component types), *assembly* (i.e., defining instance event subscriptions), *rate* (i.e., defining instance execution priority), *distribution* (i.e., defining an instance's location within the system), *dependences* (i.e., defining dependencies between component inputs and outputs), *states and transitions* (i.e., defining component attributes that persist across method executions and transitions of attributes achieved by execution of component methods and event handlers), and *synchronization* (i.e., defining synchronization policies for component methods) Our approach allows related specifications to be viewed as refinements, for example, one transition system description may refine another or it may refine a dependence specification. Using these forms, a developer may start with IDL and then selectively add focused semantic information to enable specific kinds of analysis and synthesis.

Associated with each of these specification forms is an analysis capability including: *event dependence checking* (i.e., using dependence and state transition forms to answer queries based on forward/backward slicing/chopping, and to detect anomalous event sequences such as cycles and published events without subscribers), *design advice* (i.e., heuristic-driven algorithms that use structural properties of component assemblies to generate candidate assignments for instance rate and location assignments), and *state-space search* (i.e., an abstract parameterizable semantic model of middleware and environment behavior is combined with state transition forms to answer queries about the sequencing of program actions and reachable component states). As more forms are layered onto a system description the analyses consider their composition, for example, state-space search will exploit specified or generated component rate information to eliminate searching infeasible system schedules. When a form is absent the analyses make safe assumptions about the unspecified behavior (e.g., that a component may run at any rate). Additional forms of analysis, such as timing and schedulability analysis, are being integrated into Cadena through external tool APIs.

The goal of these analyses is two-fold: to provide integrators with feedback about system properties and to drive high-quality system synthesis. Cadena is currently capable of synthesizing system configuration code that encodes event subscription, rate and distribution information for Bold-Stroke middleware. Ongoing work is enriching these capabilities to add synthesis of component code from state transition and synchronization policy specifications. This holds the promise of further simplifying component development by reducing it to straight-line sequential code with calls to well-understood library routines to achieve application specific data transformation. Future work includes the definition of customization APIs in middleware frameworks that enable model-driven optimization of execution paths within the middleware. While individual examples of such optimizations, for example replacing publish-subscribe mechanisms with method calls for co-located components, have proven to be very effective, we are working towards more general support for middleware customization.

## 3. CURRENT STATUS AND ONGOING WORK

Cadena is under active development, but we are making binary releases available to community at `http://cadena.projects.cis.ksu.edu`. Releases include a tutorial and a variety of example system models. A wide variety of system description and visualization forms are currently supported as are a suite of analyses. Support for system generation is via integration with OpenCCM.

Work on Cadena is supported by the U.S. Army Research Office (DAAD190110564) and by DARPA/IXO's PCES program (AFRL Contract F33615-00-C-3044). As part of these efforts, we are applying Cadena to model, analyze and generate systems that are representative of actual mission computing systems for fighter aircraft in terms of both their size (more than six hundred components) and complexity (thousands of event publications per scheduling period).

## 4. REFERENCES

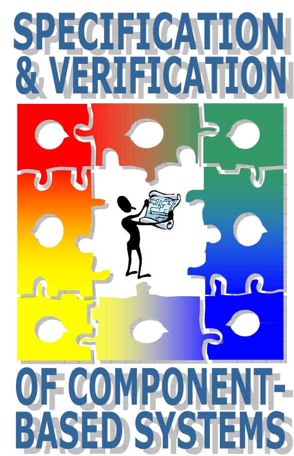

[1] B. Doerr and D. Sharp. Freeing product line architectures from execution dependencies. In *Proceedings of the Software Technology Conference*, May 1999.

[2] GOAL. The OpenCCM platform. `http://corbaweb.lifl.fr/OpenCCM/`, 2002.

[3] J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. Prasad. Cadena: An integrated development, analysis, and verification environment for component-based systems. In *Proceedings of the 25th International Conference on Software Engineering*, 2003.

[4] D. Sharp. Object oriented avionics software flow policies. In *Proceedings of the 18th AIAA/IEEE Digital Avionics Systems Conference*, Oct. 1999.

# SAVCBS 2003
# PAPERS



# Failure-free Coordinator Synthesis for Correct Components Assembly

Paola Inverardi
University of L'Aquila
Dip. Informatica
via Vetoio 1, 67100 L'Aquila, Italy
inverard@di.univaq.it

Massimo Tivoli
University of L'Aquila
Dip. Informatica
via Vetoio 1, 67100 L'Aquila, Italy
tivoli@di.univaq.it

## ABSTRACT

One of the main challenges in components assembly is related to the ability to predict possible coordination policies of the components interaction behavior by only assuming a limited knowledge of the single components computational behavior. Our answer to this problem is a software architecture based approach in which the software architecture imposed on the coordinating part of the system, allows for detection and recovery of COTS (*Commercial-Off-The-Shelf*) concurrency conflicts and for the enforcing of coordination policies on the interaction behavior of the components into composed system. Starting from the specification of the system to be assembled and of the coordination policies for the components interaction behavior, we develop a framework which automatically derives the glue code for the set of components in order to obtain a conflict-free and coordination policy-satisfying system.

## 1. INTRODUCTION

One of the main challenges in components assembly is related to the ability to predict possible coordination policies of the components interaction behavior by only assuming a limited knowledge of the single components computational behavior. Our answer to this problem is a software architecture based approach [11, 10] in which the software architecture imposed on the coordinating part of the system, allows for detection and recovery of COTS (*Commercial-Off-The-Shelf*) [17] concurrency conflicts and for the enforcing of coordination policies on the interaction behavior of the components into composed system. Building a system from a set of COTS components introduces problems related to their truly black-box nature. Since system developers have no method of looking inside the box, they can only operate on components interaction behavior to enforce coordination policies of the components into assembled system. In this context, the notion of software architecture assumes a key role since it represents the reference skeleton used to compose components and let them interact. In the software architecture domain, the interaction among the components is represented by the notion of software connector [2]. We recall that a software architecture is defined as *"the structure of the components of a system, their interrelationships, principles and guidelines governing their design and evolution over time, plus a set of connectors that mediate communication, coordination or cooperation among components."* [7].

Our approach is to compose systems by assuming a well defined architectural style [11] in such a way that it is possible to detect and to fix software anomalies. An architectural style is defined as *"a set of constraints on a software architecture that identify a class of architectures with similar features"* [4]. Moreover we assume that a specification of the desired assembled system is available and that a precise definition of the coordination policies to enforce exists. With these assumptions we are able to develop a framework that automatically derives the assembly code for a set of components so that, if possible, a conflict-free and coordination policy-satisfying system is obtained. The assembly code implements an explicit software connector (i.e. a coordinator) which mediates all interactions among the system components as a new component to be inserted in the composed system. The connector can then be analyzed and modified in such a way that the concurrency conflicts can be avoided and the specified coordination policies can be enforced on the interaction behavior of the others components into assembled system. Moreover the software architecture imposed on the composed system allows for easy replacement of a connector with another one in order to make the whole system flexible with respect to different coordination policies.

In previous works [11, 10] we limited ourselves to only concurrency conflict avoidance by enforcing only one type of coordination policy namely deadlock-free policy. In [9] we have applied the deadlock-free approach in a real scale context, namely the context of COM/DCOM applications. In this paper we generalize the framework by addressing generic coordination policies of the components into assembled system. In an other work [12] we have applied the framework we show in this paper to an instance of a typical CSCW (*Computer Supported Cooperative Work*) application, that is a collaborative writing (CW) system we have designed.

The paper is organized as follows. Sections 2 and 3 introduce background notions and, by using an explanatory example,

summarize the method concerning the synthesis of coordinators that are only deadlock-free, already developed in [11, 10]. Section 3.3 contains the main contribution of the paper and, by continuing the explanatory example, formalizes the conflict-free coordination policy-satisfying connectors synthesis. Section 4 presents related works and Section 5 discusses future work and concludes.

# 2. BACKGROUND

In this section we provide the background needed to understand the approach formalized in Section 3.

## 2.1 The reference architectural style

The architectural style we use, called *Connector Based Architecture* (CBA), consists of components and connectors which define a notion of top and bottom. The top (bottom) of a component may be connected to the bottom (top) of a single connector. Components can only communicate via connectors. Direct connection between connectors is disallowed. Components communicate synchronously by passing two type of messages: notifications and requests. A notification is sent downward, while a request is sent upward. A top-domain (bottom-domain) of a component or of a connector is the set of requests sent upward and of received notifications (of received requests received and of notifications sent downward). Connectors are responsible for the routing of messages and they exhibit a strictly sequential input-output behavior[1]. The CBA style is a generic layered style. For the sake of presentation, in this paper we describe our approach for single-layer systems. In [11] we show how to cope with multi-layered systems.

## 2.2 Configuration formalization

To our purposes we need to formalize two different ways to compose a system. The first one is called *Connector Free Architecture* (CFA) and is defined as *a set of components directly connected in a synchronous way* (i.e. without a connector). The second one is called *Connector Based Architecture* (CBA) and is defined as *a set of components directly connected in a synchronous way to one or more connectors*. In order to describe components and system behaviors we use CCS [14] (*Calculus of Communicating Systems*) notation. For the purpose of this paper this is an acceptable assumption. Actually our framework allows to automatically derive these CCS descriptions from *"HMSC (High level Message Sequence Charts)"* and *"bMSC (basic Message Sequence Charts)"* [1] specifications of the system to be assembled [16, 12]. This derivation step is performed by applying a suitable version of a translation algorithm from bMSCs and HMSCs to LTS (*Labelled Transition Systems*) [19]. HMSC and bMSC specifications are common practice in real-scale contexts thus CCS can merely be regarded as an internal to the framework specification language. Since these specifications model finite-state behaviors of a system we will use finite-state CCS:

*Definition 1. Connector Free Architecture (CFA)*: $CFA \equiv (C_1 \mid C_2 \mid ... \mid C_n)\backslash \bigcup_{i=1}^{n} Act_i$ where for all $i = 1,..,n$, $Act_i$ is the actions set of the CCS process $C_i$.

[1]Each input action is strictly followed by the corresponding output action.

*Definition 2. Connector Based Architecture (CBA)*: $CBA \equiv (C_1[f_1] \mid C_2[f_2] \mid ... \mid C_n[f_n] \mid K)\backslash \bigcup_{i=1}^{n} Act_i[f_i]$ where for all $i = 1,..,n$, $Act_i$ is the actions set of the CCS process $C_i$ and $f_i$ is a relabelling functions such that $f_i(\alpha) = \alpha_i$ for all $\alpha \in Act_i$ and $K$ is the CCS process representing the connector.

In Figure 1 we show an example of CFA system and of the corresponding CBA system. The double circled states represent initial states.

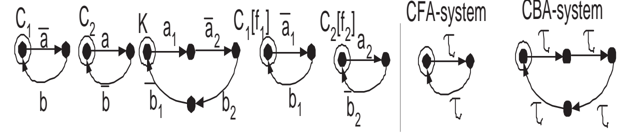



**Figure 1: CFA and corresponding CBA**

# 3. APPROACH DESCRIPTION

The problem we want to treat can be informally phrased as follows: *given a CFA system $T$ for a set of black-box interacting components, $C_i$, and a set of coordination policies $P$ automatically derive the corresponding CBA system $V$ which implements every policy in $P$.*

We are assuming that a specification of the system to be assembled is provided. Referring to Definition 1, we assume that for each component a description of its behavior as finite-state CCS term is provided (i.e. LTS *Labelled Transitions System*). Moreover we assume that a specification of the coordination policies to be enforced exists. In the following, by means of a working example, we discuss our method proceeding in three steps as illustrated in Figure 2.

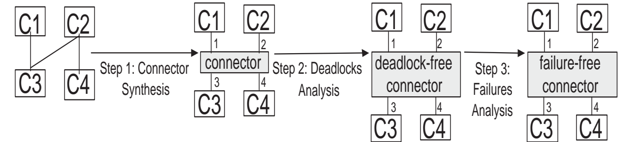



**Figure 2: 3 step method**

The first step builds a connector (i.e. the coordinator) following the CBA style constraints. The second step performs the concurrency conflicts (i.e. deadlocks) detection and recovery process. Finally, the third step performs the enforcing of the specified coordination policies against the conflict-free connector and then synthesizes a coordination policy-satisfying connector. The first two steps concern the approach already developed in our precedent works [11, 10]. Instead the third step concerns the extension of the approach to deal with generic coordination policy. From the latter we can derive the code implementing the coordinator component which is by construction correct with respect to the coordination policies specification.

Note that although in principle we could carry on the second and third step together we decided to keep them separate. Actually, the current framework implementation follows this schema.

## 3.1 First step: Coordinator Synthesis

The first step of our method (see Figure 2) starts with a CFA system and produces the equivalent CBA system. It is worthwhile noticing that this can always be done [11]. We proceed as follows:

i) for each finite-state CCS component specification in the CFA system we derive the corresponding AC-Graph. AC-Graphs model components behavior in terms of interactions with the external environment. AC-Graph carry on information on both labels and states:

*Definition 3. AC-Graph*:
Let $\langle S_i, L_i, \rightarrow_i, s_i \rangle$ be a labelled transition system of a component $C_i$. The corresponding Actual Behavior (AC) Graph $AC_i$ is a tuple of the form
$\langle N_{AC_i}, LN_{AC_i}, A_{AC_i}, LA_{AC_i}, s_i \rangle$ where $N_{AC_i} = S_i$ is a set of nodes, $LN_{AC_i}$ is a set of state labels, $LA_{AC_i}$ is a set of arc labels with $\tau$ ( $LA_{AC_i} = L_i \bigcup \tau$), $A_{AC_i} \subseteq N_{AC_i} \times LA_{AC_i} \times N_{AC_i}$ is a set of arcs and $s_i$ is the root node.

- We shall write $g \xrightarrow{l} h$, if there is an arc (g, l, h) $\in A_{AC_i}$. We shall also write $g \rightarrow h$ meaning that $g \xrightarrow{l} h$ for some l $\in LA_{AC_i}$.
- If $t = l_1 \cdots l_n \in LA^*_{AC_i}$, then we write $g \overset{t}{\longrightarrow}{}^{*} h$, if $g \xrightarrow{l_1} \cdots \xrightarrow{l_n} h$. We shall also write $g \longrightarrow^* h$, meaning that $g \overset{t}{\longrightarrow}{}^{*} h$ for some $t \in LA^*_{AC_i}$.
- We shall write $g \overset{l}{\Rightarrow} h$, if $g \overset{t}{\longrightarrow}{}^{*} h$ for some $t \in \tau^*.l.\tau^*$.

In Figure 3 we show the AC-Graphs of the CFA system of our explanatory example. The double-circled states are the initial states. For the transition labels we use a CCS notation ($\alpha$ is an input action and $\overline{\alpha}$ is the corresponding output action).

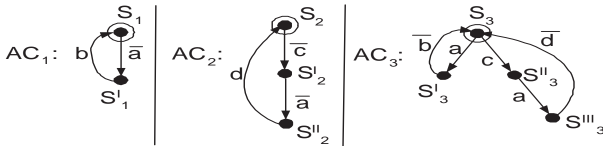



**Figure 3: AC-Graphs of the example**

We are assuming a client-server components setting. $AC_1$ and $AC_2$ are the AC-Graphs of the two client components (i.e. $C_1$ and $C_2$). $AC_3$ is the AC-Graph of the server component (i.e. $C_3$). $C_3$ exports two services, namely $a$ and $c$. $c$ has not a return value. $a$ has either $b$ or $d$ as return values. The input actions on $AC_3$ represent requests of service from the clients while the output actions represent return values towards the clients. The input actions on $AC_1$ and $AC_2$ represent return values from the server while the output actions represent requests of service towards the server.

ii) We derive from AC-Graph the requirements on its environment that guarantee concurrency conflict (i.e. deadlock) freedom. Referring to Definition 1, the environment of a component $C_i$ is represented by the set of components $C_j$ ($j \neq i$) in parallel. A component will not be in conflict with its environment if the environment can always provide the actions it requires for changing state. This is represented as AS-Graphs (Figure 4):

*Definition 4. AS-Graph*:
Let ($N_{AC_i}, LN_{AC_i}, A_{AC_i}, LA_{AC_i}, s_i$) be the AC-Graph $AC_i$ of a component $C_i$, then the corresponding ASsumption (AS) Graph $AS_i$ is ($N_{AS_i}$, $LN_{AS_i}$, $A_{AS_i}$, $LA_{AS_i}$, $s_i$) where $N_{AS_i} = N_{AC_i}$, $LN_{AS_i} = LN_{AC_i}$, $LA_{AS_i} = LA_{AC_i}$ and $A_{AS} = \{(\nu, \overline{a}, \nu') \mid (\nu, a, \nu') \in A_{AC}\} \bigcup \{(\nu, b, \nu') \mid (\nu, \overline{b}, \nu') \in A_{AC}\}$.

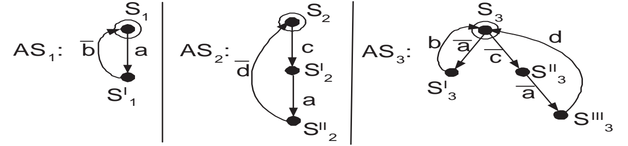



**Figure 4: AS-Graphs of the example**

Now if we consider Definition 2, the environment of a component can only be represented by connectors, EX-Graph represents the behavior that the component expects from the connectors (Figure 5):

*Definition 5. EX-Graph*: Let ($N_{AS_i}$, $LN_{AS_i}$, $A_{AS_i}$, $LA_{AS_i}$, $s_i$) be the AS-Graph $AS_i$ of a component $C_i$; we define the connector EXpected (EX) Graph $EX_i$ from the component $C_i$ the graph ($N_{EX_i}$, $LN_{EX_i}$, $A_{EX_i}$, $LA_{EX_i}$, $s_i$), where:

- $N_{EX_i} = N_{AS_i}$ and $LN_{EX_i} = LN_{AS_i}$
- $A_{EX_i}$ and $LA_{EX_i}$ are empty
- $\forall$ ($\mu$, $\alpha$, $\mu'$) $\in A_{AS_i}$, with $\alpha \neq \tau$
  - Create a new node $\mu_{new}$ with a new unique label, add the node to $N_{EX_i}$ and the unique label to $LN_{EX_i}$
  - if ($\mu$, $\alpha$, $\mu'$) is such that $\alpha$ is an input action (i.e. $\alpha$ = a, for some a )
    - add the labels $a_i$ and $\overline{a_?}$ to $LA_{EX_i}$
    - add ($\mu$, $a_i$, $\mu_{new}$) and ($\mu_{new}$, $\overline{a_?}$, $\mu'$) to $A_{EX_i}$
  - if ($\mu$, $\alpha$, $\mu'$) is such that $\alpha$ is an output action (i.e. $\alpha = \overline{a}$, for some a )
    - add the labels $\overline{a_i}$ and $a_?$ to $LA_{EX_i}$
    - add ($\mu$, $a_?$, $\mu_{new}$) and ($\mu_{new}$, $\overline{a_i}$, $\mu'$) to $A_{EX_i}$
- $\forall$ ($\mu$, $\tau$, $\mu'$) $\in A_{AS_i}$ add $\tau$ to $LA_{EX_i}$ and ($\mu$, $\tau$, $\mu'$) to $A_{EX_i}$

iii) Each EX-Graph represents a partial view (i.e. the single component's view) of the connector behavior. The EX-Graph for component $C_i$ (i.e. $EX_i$) is the behavior that $C_i$ expects from the connector. Thus $EX_i$ has either transitions

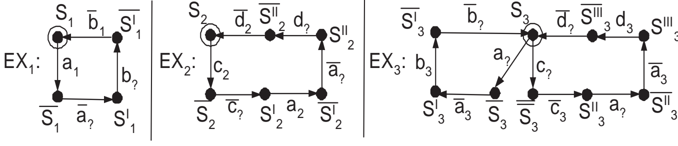



**Figure 5: EX-Graphs of the example**

labelled with known actions or with unknown actions for $C_i$. Known actions are performed on the channel connecting $C_i$ to the connector. This channel is known to $C_i$ and identified by a number. Unknown actions are performed on channels connecting other components $C_j$ $(j \neq i)$ to the connector, therefore unknown from the $C_i$ perspective. These channels are identified by the question mark. We derive the connector global behavior through the following EX-Graphs unification algorithm.

*Definition 6. EX-Graphs Unification*:

- Let $C_1, .., C_n$ be the components in CFA-version of the composed system in such a way that $\{C_1, .., C_h\}$ is the set of null bottom domain components and $\{C_{h+1}, .., C_n\}$ is the set of null top domain components;
- Let $EX_1, .., EX_h, EX_{h+1}, .., EX_n$ be their corresponding EX-Graphs;
- Let $1, .., h, h+1, .., n$ be their corresponding communication channels;
- Let $S_1, .., S_h, S_h+1, .., S_n$ be their corresponding current states.

At the beginning the current states are the initial states.

1. Create the actual behavior graph of the connector, with one node (initial state) and no arcs.
2. Set as current states of the components $EX-Graphs$ the respective initial states.
3. Label the connector initial state with an ordered tuple composed of the initial states of all components (null bottom domain and null top domain). For simplicity of presentation we assume to order them so that the $j-th$ element of the state label corresponds to the current state of the component $C_j$ where $j \in [1, .., h, h+1, .., n]$. This state is set as the connector current state.
4. Perform the following unification procedure:
   (a) Let $g$ be the connector current state. Mark $g$ as visited.
   (b) Let $< S_1, .., S_h, S_{h+1}, .., S_n >$ be the state label of $g$.
   (c) Generate the set $TER$ of action_terms and the set $VAR$ of action_variables so that $t_i \in TER$, if in $EX_i$ $S_i \stackrel{t_i}{\Rightarrow} \overline{S}_i$. Similarly $v_? \in VAR$, if $\exists$ $j$ in such a way that in $EX_j$ $S_j \stackrel{v_?}{\Rightarrow} \overline{S}_j$.
   (d) For all unifiable pairs $(t_i,\ v_?)$, with $i \neq j$ do:
      i. if $i \in \{1, .., h\}$, $j \in \{h+1, .., n\}$ and they do not already exist then create new nodes (in the connector graph) $g_i, g_j$ with state label $< S_1, .., \overline{S}_i, .., S_h, S_{h+1}, .., \overline{S}_j, .., S_n >$ and $< S_1, .., S'_i, .., S_h, S_{h+1}, .., S'_j, .., S_n >$ respectively,
      where in $AS_i$ $S_i \stackrel{t}{\Rightarrow} S'_i$ and in $AS_j$ $S_j \stackrel{\overline{v}}{\Rightarrow} S'_j$;
      ii. if $j \in \{1, .., h\}$, $i \in \{h+1, .., n\}$ and they do not already exist then create new nodes (in the connector graph) $g_i$, $g_j$ with state label $< S_1, .., \overline{S}_j, .., S_h, S_{h+1}, .., \overline{S}_i, .., S_n >$ and $< S_1, .., S'_j, .., S_h, S_{h+1}, .., S'_i, .., S_n >$ respectively,
      where in $AS_i$ $S_i \stackrel{t}{\Rightarrow} S'_i$ and in $AS_j$ $S_j \stackrel{\overline{v}}{\Rightarrow} S'_j$;
      iii. create the arc $(g,\ t_i,\ g_i)$ in the connector graph;
      iv. mark $g_i$ as visited;
      v. create the arc $(g_i,\ \overline{v}_j,\ g_j)$ in the connector graph.
   (e) Perform recursively this procedure on all not marked (as visited) adjacent nodes of current node.

In Figure 6 we show the connector graph for the example illustrated in this section. The resulting CBA system is built as defined in Definition 2.

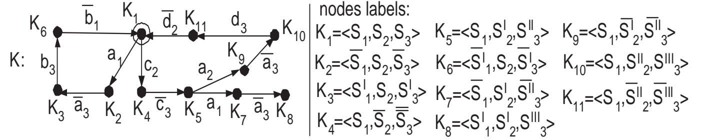



**Figure 6: Connector graph of the example**

In [11] we have proved that the CBA-system obtained by the connector synthesis process is equivalent to the corresponding CFA-system. To do this we have proved that the CFA-system can be simulated by the synthesized CBA-system (correctness of the synthesis) under a suitable notion of *"state based"*[2] equivalence called CB-Simulation [11]. The starting point of CB-Simulation is the stuttering equivalence [15]. We have also proved that the connector does not introduce in the system any new logic (completeness of the synthesis).

### 3.2 Second step: Concurrency conflicts avoidance

The second step concerns the concurrency conflicts avoidance, which is performed on the CBA system. In [11], we have proved that if a concurrency conflict (i.e. coordination deadlock) is possible, then this results in a precise connector behavior that is detectable by observing the connector graph. To fix this problem it is enough to prune all the finite branches of the connector transition graph. The pruned connector preserves all the correct (with respect to deadlock freeness) behaviors of CFA-system [11]. In Figure 7 we show the concurrency conflict-free connector graph.

[2]By definition, both CFA and CBA systems exhibit only $\tau$ transitions.

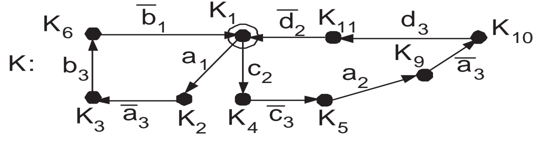



**Figure 7: Deadlock-free connector graph of the example**

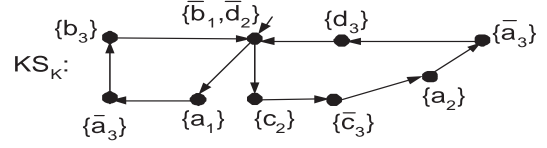



**Figure 8: Kripke structure of $K$**

## 3.3 Third step: Coordination policies enforcing

In this section we formalize the third step of the method of Figure 2. This step concerns the coordination policy enforcing on the connector graph.

### 3.3.1 *Generic coordination policies specification:*

The behavioral properties we want to enforce are related to behaviors of the CFA system that concern coordination policies of the interaction behavior of the components in the CFA system. The CFA behaviors that do not comply to the specified properties represent behavioral failures. A behavior of the CFA system is given in terms of sequences of actions performed by components in the CFA system. In specifying properties we have to distinguish an action $\alpha$ performed by a component $C_i$ with the same action $\alpha$ performed by a component $C_j$ $(i \neq j)$. Thus, referring to Definition 1, the behavioral properties (i.e. coordination properties) can only be specified in terms of visible actions of the components $C_1[f_1], C_2[f_2], .., C_n[f_n]$ where for each $i = 1, .., n,\ f_i$ is a relabelling function such that $f_i(\alpha) = \alpha_i$ for all $\alpha \in\ Act_i$ and $Act_i$ is the actions set for $C_i$. By referring to the usual model checking approach [5] we specify every property through a temporal logic formalism. We choose $LTL$ [5] (*Linerar-time Temporal Logic*) as specification language. We define $AP = \{\gamma\ :\ \gamma = l_i\ \vee\ \gamma = \overline{l}_i\ with\ l \in\ LA_{AC_i}, l \neq \tau, i = 1, .., n\}$ as the set of atomic proposition on which we define the LTL formulas corresponding to the coordination policies. Refer to [5] for the standard LTL syntax and semantics.

### 3.3.2 *Enforcing a coordination policy:*

The semantics of a LTL formula is defined with respect to a model represented by a Kripke structure. We consider as Kripke structure corresponding to the connector graph $K$ a connector model $KS_K$ that represents the Kripke structure of $K$. $KS_K$ is defined as follows:

*Definition 7. Kripke structure of a connector graph K:* Let $(N, LN, LA, A, k_1)$ be the connector graph $K$. We define the Kripke Structure of $K$, the Kripke structure $KS_K = (N, A, \{k_1\}, LV)$ where $LV \subseteq\ 2^{LA}$ with $LV(k_1) = \{\alpha_i\ :\ LA((\overline{k}, k_1)) = \alpha_i,\ (\overline{k}, k_1) \in\ A\}$. For each $v \in\ N$ then $LV(v)$ is interpreted as the set of atomic propositions true in state $v$.

In Figure 8, we show the Kripke structure of $K$. The node with an incoming little-arrow is the initial state. In Section 3.3.1 we have described how we can specify a property in terms of desired CFA behaviors. We have also said that all the undesired behaviors represent CFA failures. Analogously to deadlocks analysis, we can solve behavioral failures of the CFA system that are identifiable in the corresponding CBA system with precise behaviors of the synthesized connector. A connector behavior is simply an execution path into the connector graph. An execution path is a sequence of state's transition labels. It is worthwhile noticing that the behavioral properties (i.e. coordination properties) that we specify for the CFA system are corresponding to behavioral properties of the connector in the CBA system. In fact every action $\gamma = \alpha_i \in AP$ can be seen as the action $\overline{\alpha}$ (into the connector graph) performed on the communication channel that connects $C_i$ to the connector. This is true for construction (see Section 3.1). Thus let $P$ be a behavioral property specification (i.e. LTL formula) for the CFA system, we can translate $P$ in another behavioral property: $P_{cba}$. $P_{cba}$ is automatically obtained by applying the CCS complement operator to the atomic propositions in $P$. $P_{cba}$ is the property specification for the CBA system corresponding to $P$. Then we translate $P_{cba}$ in the corresponding Büchi Automaton [5] $B_{P_{cba}}$:

*Definition 8. Büchi Automaton:* A Büchi Automaton $B$ is a 5-tuple $< S, A, \triangle, q_0, F >$, where $S$ is a finite set of states, $A$ is a set of actions, $\triangle \subseteq\ S\times\ A\times\ S$ is a set of transitions , $q_0 \in\ S$ is the initial state, and $F \subseteq\ S$ is a set of accepting states. An *execution* of $B$ on an infinite word $w = a_0a_1...$ over $A$ is an infinite sequence $\sigma\ = q_0q_1...$ of elements of $S$, where $(q_i, a_i, q_{i+1}) \in\ \triangle,\ \forall\ i\ \geq\ \ 0$. An execution of $B$ is *accepting* if it contains some accepting state of $B$ an infinite number of times. $B$ accepts a word $w$ if there exists an accepting execution of $B$ on $w$.

Referring to our example we consider the following behavioral property: $P = F((\overline{a}_1 \wedge X(!\overline{a}_1 U \overline{a}_2)) \vee (\overline{a}_2 \wedge X(!\overline{a}_2 U \overline{a}_1)))$. This property specifies all CFA system behaviors that guarantee the evolution of all components in the system. It specifies that the components $C_1$ and $C_2$ can perform the action $a$ by necessarily using an alternating coordination policy. In other words it means that if the component $C_1$ performs an action $a$ then $C_1$ cannot perform $a$ again if $C_2$ has not performed $a$ and viceversa. The connector to be synthesized will avoid starvation by satisfying this property. In Figure 9 we show $B_{P_{cba}}$. We recall that $P_{cba} = F((a_1 \wedge X(!a_1 U a_2)) \vee (a_2 \wedge X(!a_2 U a_1)))$; $p_0$ and $p_2$ are the initial and accepting state respectively.

Given a Büchi Automaton $A$, $L(A)$ is the *language* consisting of all words accepted by $A$. Moreover to a Kripke structure $T$ corresponds a Büchi Automaton $B_T$ [5]. We can derive $B_{KS_K}$ as the Büchi Automaton corresponding to $KS_K$ (see Figure 9). The double-circled states are accepting states. Given $B_{KS_K} = (N, A', \Delta, \{s\}, N)$ and $B_P =$

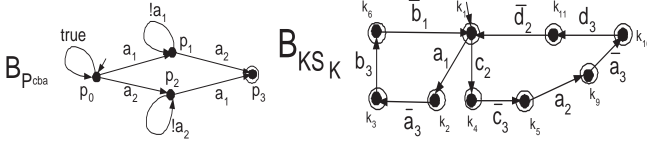



**Figure 9: Büchi Automata $B_{P_{cba}}$ and $B_{KS_K}$ of $P_{cba}$ and $KS_K$ respectively**

$(S, A'', \Gamma, \{v\}, F)$ the method performs the following enforcing procedure in order to synthesize a deadlock-free connector graph that satisfies the property $P$:

1. *build the automaton that accepts* $L(B_{KS_K}) \cap L(B_{P_{cba}})$*; this automaton is defined as* $B^{K,P}_{intersection} = (S \times N, A', \Delta', \{< v, s >\}, F \times N)$ *where* $(< r_i, q_j >, a, < r_m, q_n >) \in \Delta'$ *if and only if* $(r_i, a, r_m) \in \Gamma$ *and* $(q_j, a, q_n) \in \Delta$*;*

2. *if* $B^{K,P_{cba}}_{intersection}$ *is not empty return* $B^{K,P_{cba}}_{intersection}$ *as the Büchi Automaton corresponding to the* $P$*-satisfying execution paths of* $K$*.*

In Figure 10, we show $B^{K,P_{cba}}_{intersection}$.

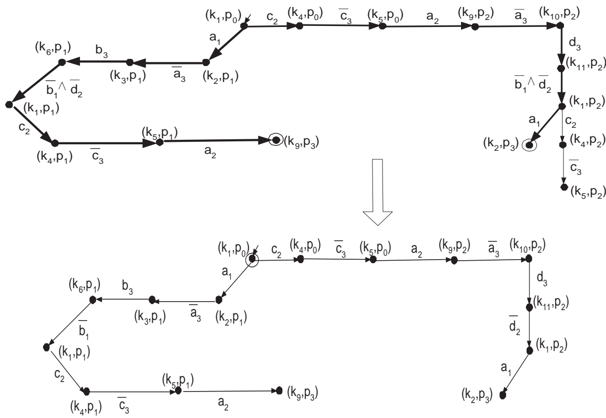



**Figure 10: $B^{K,P_{cba}}_{intersection}$ and deadlock-free property-satisfying connector graph of the explanatory example**

Finally our method derives from $B^{K,P_{cba}}_{intersection}$ the corresponding connector graph. This graph is constructed by considering the execution paths of $B^{K,P_{cba}}_{intersection}$ that are only *accepting* (see the path made of bold arrows in Figure 10); we define an *accepting execution path* of $B^{K,P_{cba}}_{intersection}$ as follows:

*Definition 9. Accepting execution path of* $B^{K,P_{cba}}_{intersection}$*:* Let $B^{K,P_{cba}}_{intersection} = (S \times N, \Delta', \{< v, s >\}, F \times N)$ be the automaton that accepts $L(B_{KS_K}) \cap L(B_{P_{cba}})$. We define an accepting execution path of $B^{K,P_{cba}}_{intersection}$ a sequence of states $\gamma = s_1, s_2, .., s_n$ such that $\forall\, i = 1, .., n \;:\; s_i \in S \times N$; for $1 \leq i \leq n-1$, $(s_i, s_{i+1}) \in \Delta'$ and $(s_n, s_1) \in \Delta'$ or $(s_n, s_1) \notin \Delta'$; and $\exists\, k = 1, .., n \;:\; k \in F \times N$.

In Figure 10, we show the deadlock-free property-satisfying connector graph for our explanatory example. Depending on the property, this graph could contain finite paths (i.e. paths terminating with a stop node). Note that at this stage the stop nodes representing accepting states. In fact we have obtained the deadlock-free property-satisfying connector graph by considering only the accepting execution paths of $B^{K,P_{cba}}_{intersection}$, thus stop nodes represent connector states satisfying the property. Once the connector has reached an accepting stop node it will return to its initial state waiting for a new request from an its client. Returning to the initial state is not explicitly represented in the deadlock-free property-satisfying connector graph but it will be implicitly considered in the automatic derivation of the code implementing the deadlock-free property-satisfying connector. By visiting this graph and by exploiting the information stored in its states and transitions we can automatically derive the code that implements the P-satisfying deadlock-free connector (i.e. the coordinator component) analogously to what done for deadlock-free connectors [10]. The implementation refers to Microsoft COM (*Component Object Model*) components and uses C++ wiht ATL (*Active Template Library*) as programming environment. The connector component $K$ implements the COM interface $IC3$ of the component $C_3$ by defining a COM class $K$ and by implementing a wrapping mechanism in order to wrap the requests that $C_1$ and $C_2$ perform on component $C_3$ (actions $\overline{a}$ and $\overline{c}$ on $AC_1$ and $AC_2$ of Figure 3). In the following we show fragments of the IDL (*Interface Definition Language*) definition for $K$, of the $K$ COM library and of the $K$ COM class respectively. $c3Obj$ is an instance of the inner COM server corresponding to $C_3$ and encapsulated into connector component $K$.

```
import ic3.idl; ... library K_Lib {
   ...
   coclass K {
      [default] interface IC3;
   }
}

...

class K : public IC3 {
   // stores the current state of the connector
   private static int sLbl;

   // stores the current state of the
   // property automaton
   private static int pState;

   // stores the number of clients
   private static int clientsCounter = 0;

   // channel's number of a client
   private int chId;

   // COM smart pointer; is a reference to
   // the C3 server object
   private static C3* c3Obj;

   ...

   // the constructor
   K() {
      sLbl = 1;
      pState = 0;
      clientsCounter++;
      chId = clientsCounter;
      c3Obj = new C3();
      ...
   }

   // implemented methods
   ...
}
```

In the following we show the deadlock-free property-satisfying code implementing the methods ***a*** and ***c*** of the connector component $K$. Even if the property $P$ of our example considers a coordination policy only for action $a$, we have to

coordinate also the requests of $c$ in order to satisfy $P$. Actually, as we can see in Figure 10, the deadlock-free property-satisfying connector has execution paths in which transitions labelled with $c$ there exist.

```
HRESULT a(/* params list of a */) {
   if(sLbl == 1) {
      if((chId == 1) && (pState == 0)) {
         return c3Obj->a(/* params list of a */);
         pState = 1; sLbl = 1; //it goes on the state preceding the next
                               //request of a method from a client
      }
      else if((chId == 1) && (pState == 2)) {
         return c3Obj->a(/* params list of a */);
         pState = 0; sLbl = 1; //since it has found an accepting stop node,
                               //it returns to its initial state
      }
   }
   else if(sLbl == 5) {
      if((chId == 2) && (pState == 1)) {
         return c3Obj->a(/* params list of a */);
         pState = 0; sLbl = 1; //since it has found an accepting stop node,
                               //it returns to its initial state
      }
      else if((chId == 2) && (pState == 0)) {
         return c3Obj->a(/* params list of a */);
         pState = 2; sLbl = 1; //it goes on the state preceding the next
                               //request of a method from a client
      }
   }

   return E_HANDLE;
}
```

```
HRESULT c(/* params list of c */) {
    if(sLbl == 1) {
      if((chId == 2) && (pState == 1)) {
         return c3Obj->a(/* params list of a */);
         pState = 1; sLbl = 5; //it goes on the state preceding the next
                               //request of a method from a client
      }
      else if((chId == 2) && (pState == 0)) {
         return c3Obj->a(/* params list of a */);
         pState = 0; sLbl = 5; //it goes on the state preceding the next
                               //request of a method from a client
      }
   }

   return E_HANDLE;
}
```

In [13] we prove the correctness of the property enforcing procedure. We prove that the CBA-system based on the property-satisfying deadlock-free connector preserves all the property-satisfying behaviors of the corresponding deadlock-free CFA-system.

# 4. RELATED WORKS

The architectural approach to correct and automatic connector synthesis presented in this paper is related to a large number of other problems that have been considered by researchers over the past two decades. For the sake of brevity we mention below only the works closest to our approach. The most strictly related approaches are in the "*scheduler synthesis*" research area. In the discrete event domain they appear as "*supervisory control*" problem [3, 18]. In very general terms, these works can be seen as an instance of a problem similar to the problem treated in our approach. However the application domain of these approaches is sensibly different from the software component domain. Dealing with software components introduces a number of problematic dimensions to the original synthesis problem. There are two main problems with this approach: i) the computational complexity and the state-space explosion and ii) in general the approach is not compositional. The first problem can be avoided by using a logical encoding of the system specification in order to use a more efficient data structure (i. e. BDD (Binary Decision Diagram)) to perform the supervisor synthesis; however the second problem cannot be avoided and only under particular conditions it is possible to synthesize the global complete supervisor by composing modular supervisors. While the state-space explosion is a problem also present in our approach, on the other side we have proved in [11] that our approach is always compositional. It means that if we build the connector for a given set of components and later we add a new component in the resulting system we can extend the already available connector and we must not perform again the entire synthesis process.

Other works that are related to our approach, appear in the *model checking of software components* context in which CRA (*Compositional Reachability Analysis*) techniques are largely used [8]. Also these works can be seen as an instance of the general problem formulated in Section 3. They provide an optimistic approach to software components model checking. These approaches suffer the state-space explosion problem. However this problem is raised only in the worst case that may not be the case often in practice. In these approaches the assumptions that represent the *weakest* environment in which the components satisfy the specified properties are automatically synthesized. However the synthesized environment does not provide a model for the properties satisfying glue code. The synthesized environment may be rather used for runtime monitoring or for components retrieval.

Recently promising formal techniques for the compositional analysis of component based design have been developed [6]. The key of these works is the modular-based reasoning that provides a support for the modular checking of behavioral properties. The goal of these works is quite different from our in fact they are related only to software components interfaces compatibility check. Thus they provide only a check on component-based design level.

# 5. CONCLUSION AND FUTURE WORKS

In this paper we have described a connector-based architectural approach to component assembly. Our approach focusses on detection and recovery of the assembly behavioral failures. A key role is played by the software architecture structure since it allows all the interactions among components to be explicitly routed through a synthesized connector. We have applied our approach to an example and we have discussed its implications on the actual nature of black-box components. As far as components are concerned we only assumed to have a CCS description of the components behavior. For the purpose of this paper this is an acceptable assumption. However our framework allows to automatically derive these CCS descriptions from specifications that are common practice in real-scale contexts. For behavioral properties we have shown in this paper how to go beyond deadlock. The complexity of the synthesis and analysis algorithm is exponential either in space and time. This value of complexity is obtained by considering the unification process complexity and the size of the data structure used to build the connector graph. At present we are studying better data structures for the connector model in order to reduce their size. By referring to the automata based model checking [5], we are also working to perform on the fly analysis during the connector model building process. Other possible limits of the approach are: i) we completely

centralize the connector logic and we provide a strategy for the connector source code derivation step that derives a centralized implementation of the connector component. We do not think this is a real limit because even if we centralize the connector logic we can actually think of deriving a distributed implementation of the connector component; ii) we assume that an HMSC and bMSC specification for the system to be assembled is provided. Although this is reasonable to be expected, it is interesting to investigate testing and inspection techniques to directly derive from a COTS (black-box) component some kind (possibly partial) behavioral specification; iii) we assume also an LTL specification for the behavioral property to be checked. It is interesting to find a more user-friendly property specification; for example by extending the HMSC and bMSC notations to express more complex system's components interaction behaviors.

## Acknowledgements

This work has been partially supported by Progetto MIUR SAHARA.

## 6. REFERENCES

[1] Itu telecommunication standardisation sector, itu-t reccomendation z.120. message sequence charts. (msc'96). Geneva 1996.

[2] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions On Software Engineering and Methodology, Vol. 6, No. 3, pp. 213-249*, 6(3):213–249, July 1997.

[3] S. Balemi, G. J. Hoffmann, P. Gyugyi, H. Wong-Toi, and G. F. Franklin. Supervisory control of a rapid thermal multiprocessor. *IEEE Transactions on Automatic Control*, 38(7):1040–1059, July 1993.

[4] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison Wesley, 1998.

[5] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, London, England, 2001.

[6] L. de Alfaro and T. Heinzinger. Interface automata. In *ACM Proc. of the joint 8th ESEC and 9th FSE, ACM Press*, Sep 2001.

[7] D. Garlan and D. E. Perry. *Introduction to the Special Issue on Software Architecture, Vol. 21. Num. 4. pp. 269-274*, April 1995.

[8] D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. Assumption generation for software component verification. *Proc. 17th IEEE Int. Conf. Automated Software Engineering 2002*, September 2002.

[9] P. Inverardi and M. Tivoli. Deadlock-free software architectures for com/dcom applications. *Journal of Systems and Software, Volume 65, Issue 3, 15 March 2003, Pages 173-183, Component-Based Software Engineering.*

[10] P. Inverardi and M. Tivoli. Automatic synthesis of deadlock free connectors for com/dcom applications. In *ACM Proceedings of the joint 8th ESEC and 9th FSE, ACM Press*, Vienna, Sep 2001.

[11] P. Inverardi and M. Tivoli. Connectors synthesis for failures-free component based architectures. *Technical Report, University of L'Aquila, Department of Computer Science, http://sahara.di.univaq.it/tech.php?id_tech=7 or http://www.di.univaq.it/~tivoli/ffsynthesis.pdf*, ITALY, January 2003.

[12] P. Inverardi, M. Tivoli, and A. Bucchiarone. Automatic synthesis of coordinators of cots group-ware applications: an example. *International Workshop on Distributed and Mobile Collaboration (DMC 2003), http://www.di.univaq.it/tivoli/Publications /Full/DMC2003.pdf, 9-11 June, Linz, Austria WETICE 2003.*

[13] P. Inverardi, M. Tivoli, and A. Bucchiarone. Failures-free connector synthesis for correct components assembly. *Technical Report, University of L'Aquila, Department of Computer Science, http://www.di.univaq.it/tivoli/ffs_techrep.pdf*, ITALY, March 2003.

[14] R. Milner. *Communication and Concurrency*. Prentice Hall, New York, 1989.

[15] R. D. Nicola and F. Vaandrager. Three logics for branching bisimulation. *Journal of the ACM*, 42(2):458–487, 1995.

[16] P.Inverardi and M.Tivoli. Automatic failures-free connector synthesis: An example. *published on the post-workshop proceedings of Monterey 2002 Workshop: Radical Innovations of Software and Systems Engineering in the Future, Venezia (ITALY)*, September 2002.

[17] C. Szyperski. *Component Software. Beyond Object Oriented Programming*. Addison Wesley, Harlow, England, 1998.

[18] E. Tronci. Automatic synthesis of controllers from formal specifications. *Proc. of 2nd IEEE Int. Conf. on Formal Engineering Methods*, December 1998.

[19] S. Uchitel, J. Kramer, and J. Magee. Detecting implied scenarios in message sequence chart specifications. In *ACM Proceedings of the joint 8th ESEC and 9th FSE*, Vienna, Sep 2001.

# Proof Rules for Automated Compositional Verification through Learning

Howard Barringer [*]
Department of Computer Science
University of Manchester
Oxford Road, Manchester, M13 9PL
howard@cs.man.ac.uk

Dimitra Giannakopoulou
RIACS/USRA
NASA Ames Research Center
Moffett Field, CA 94035-1000, USA
dimitra@email.arc.nasa.gov

Corina S. Păsăreanu
Kestrel Technology LLC
NASA Ames Research Center
Moffett Field, CA 94035-1000, USA
pcorina@email.arc.nasa.gov

## ABSTRACT

Compositional proof systems not only enable the stepwise development of concurrent processes but also provide a basis to alleviate the state explosion problem associated with model checking. An assume-guarantee style of specification and reasoning has long been advocated to achieve compositionality. However, this style of reasoning is often non-trivial, typically requiring human input to determine appropriate assumptions. In this paper, we present novel assume-guarantee rules in the setting of finite labelled transition systems with blocking communication. We show how these rules can be applied in an iterative and fully automated fashion within a framework based on learning.

## Keywords

Parallel Composition, Automated Verification, Assumption Generation, Learning

## 1. INTRODUCTION

Our work is motivated by an ongoing project at NASA Ames Research Center on the application of model checking to the verification of autonomous software. Autonomous software involves complex concurrent behaviors for reacting to external stimuli without human intervention. Extensive verification is a pre-requisite for the deployment of missions that involve autonomy.

Given a finite model of a system and of a required property, model checking can be used to determine automatically whether the property is satisfied by the system. The limitation of this approach, commonly referred to as the "state-explosion" problem [7], is that it needs to store the explored system states in memory, which may be prohibitively large for realistic systems.

Compositional verification presents a promising way of addressing state explosion. It advocates a "divide and conquer" approach where properties of the system are decomposed into properties of its components, so that if each component satisfies its respective property, then so does the entire system. Components are therefore model checked separately. It is often the case, however, that components only satisfy properties in specific contexts (also called environments). This has given rise to the application of assume-guarantee reasoning [16, 21] to model checking [11].

Assume-guarantee[1] reasoning first checks whether a component $M$ guarantees a property $P$, when it is part of a system that satisfies an assumption $A$. Intuitively, $A$ characterizes all contexts in which the component is expected to operate correctly. To complete the proof, it must also be shown that the remaining components in the system, i.e., $M$'s environment, satisfy $A$. Several frameworks have been proposed [16, 21, 6, 14, 24, 15] to support this style of reasoning. However, their practical impact has been limited because they require non-trivial human input in defining assumptions that are strong enough to eliminate false violations, but that also reflect appropriately the remaining system.

In previous work [8], we developed a novel framework to perform assume-guarantee reasoning in an *iterative* and *fully automatic* fashion; the approach uses learning and model-checking. To check that a system made up of components $M_1$ and $M_2$ satisfies a property $P$, our framework automatically learns and refines assumptions for one of the components to satisfy $P$, which it then tries to discharge on the other component. Our approach is guaranteed to terminate, stating that the property holds for the system, or returning a counterexample if the property is violated.

This work introduces a variety of sound and complete assume-guarantee rules in the setting of Labeled Transition Systems with blocking communication. The rules are motivated by the need for automating assume-guarantee reasoning. How-

[*]This author is most grateful to RIACS/USRA and the UK's EPSRC under grant GR/S40435/01 for the partial support provided to conduct this research

[1]The original terminology for this style of reasoning was rely-guarantee or assumption-commitment; it was introduced for enabling top-down development of concurrent systems.

ever, in contrast to our previous work, they are symmetric, meaning that they are based on establishing and discharging assumptions for both components at the same time. The remainder of this paper is organized as follows. We first provide some background in Section 2, followed by some basic compositional proof rules in Section 3. The framework that automates these rules is presented in Section 4. Section 5 introduces rules that optimize and extend the basic rules. Finally, Section 6 presents related work and Section 7 concludes the paper.

# 2. BACKGROUND

We use Labeled Transition Systems (LTSs) to model the behavior of communicating components in a concurrent system. In this section, we provide background on LTSs and their associated operators, and also present how properties are expressed and checked in our framework. We also summarize the learning algorithm that is used to automate our compositional verification approach.

## 2.1 Labeled Transition Systems

Let $\mathcal{A}ct$ be the universal set of observable actions and let $\tau$ denote a local action *unobservable* to a component's environment. An LTS $M$ is a quadruple $\langle Q, \alpha M, \delta, q0\rangle$ where:

- $Q$ is a non-empty finite set of states
- $\alpha M \subseteq \mathcal{A}ct$ is a finite set of observable actions called the *alphabet* of $M$
- $\delta \subseteq Q \times \alpha M \cup \{\tau\} \times Q$ is a transition relation
- $q0 \in Q$ is the initial state

An LTS $M = \langle Q, \alpha M, \delta, q0\rangle$ is *non-deterministic* if it contains $\tau$-transitions or if $\exists (q, a, q'), (q, a, q'') \in \delta$ such that $q' \neq q''$. Otherwise, $M$ is *deterministic*.

**Traces.** A *trace* $t$ of an LTS $M$ is a sequence of observable actions that $M$ can perform starting at its initial state. For $\Sigma \subseteq \mathcal{A}ct$, we use $t{\upharpoonright}\Sigma$ to denote the trace obtained by removing from $t$ all occurrences of actions $a \notin \Sigma$. The set of all traces of $M$ is called the *language* of $M$, denoted $\mathcal{L}(M)$. We will freely use the expression "a word $t$ is accepted by $M$" to mean that $t \in \mathcal{L}(M)$. Note that the empty word is accepted by any LTS.

**Parallel Composition.** Let $M = \langle Q, \alpha M, \delta, q0\rangle$ and $M' = \langle Q', \alpha M', \delta', q0'\rangle$. We say that $M$ *transits* into $M'$ with action $a$, denoted $M \xrightarrow{a} M'$, if and only if $(q0, a, q0') \in \delta$ and $\alpha M = \alpha M'$ and $\delta = \delta'$.

The parallel composition operator $\|$ is a commutative and associative operator that combines the behavior of two components by synchronizing the actions common to their alphabets and interleaving the remaining actions.

Let $M_1 = \langle Q_1, \alpha M_1, \delta_1, q0_1\rangle$ and $M_2 = \langle Q_2, \alpha M_2, \delta_2, q0_2\rangle$ be two LTSs. Then $M_1 \| M_2$ is an LTS $M = \langle Q, \alpha M, \delta, q0\rangle$, where $Q = Q_1 \times Q_2$, $q0 = (q0_1, q0_2)$, $\alpha M = \alpha M_1 \cup \alpha M_2$, and $\delta$ is defined as follows, where $a$ is either an observable action or $\tau$ (note that the symmetric rules are implied by the fact that the operator is commutative):

$$\frac{M_1 \xrightarrow{a} M_1',\ a \notin \alpha M_2}{M_1 \| M_2 \xrightarrow{a} M_1' \| M_2}$$

$$\frac{M_1 \xrightarrow{a} M_1',\ M_2 \xrightarrow{a} M_2',\ a \neq \tau}{M_1 \| M_2 \xrightarrow{a} M_1' \| M_2'}$$

**Note.** $\mathcal{L}(M_1 \| M_2) = \{t \mid t{\upharpoonright}\alpha M_1 \in \mathcal{L}(M_1) \land t{\upharpoonright}\alpha M_2 \in \mathcal{L}(M_2) \land t \in (\alpha M_1 \cup \alpha M_2)^*\}$

**Properties and Satisfiability.** A property is also defined as an LTS $P$, whose language $\mathcal{L}(P)$ defines the set of acceptable behaviors over $\alpha P$. An LTS $M$ satisfies $P$, denoted as $M \models P$, if and only if $\forall t \in \mathcal{L}(M).t{\upharpoonright}\alpha P \in \mathcal{L}(P)$.

## 2.2 LTSs and Finite-State Machines

As will be described in section 4, our proof-rules require the use of the "complement" of an LTS. LTSs are not closed under complementation (their languages are prefix-closed), so we need to define here a more general class of finite-state machines (FSMs) and associated operators for our framework.

An FSM $M$ is a five tuple $\langle Q, \alpha M, \delta, q0, F\rangle$ where $Q, \alpha M, \delta$, and $q0$ are defined as for LTSs, and $F \subseteq Q$ is a set of accepting states.

For an FSM $M$ and a word $t$, we use $\hat{\delta}(q, t)$ to denote the set of states that $M$ can reach after reading $t$ starting at state $q$. A word $t$ is said to be *accepted* by an FSM $M = \langle Q, \alpha M, \delta, q0, F\rangle$ if $\hat{\delta}(q0, t) \cap F \neq \emptyset$. Note that in the following sections, the term trace is often used to denote a word. The *language accepted by* $M$, denoted $\mathcal{L}(M)$ is the set $\{t \mid \hat{\delta}(q0, t) \cap F \neq \emptyset\}$.

For an FSM $M = \langle Q, \alpha M, \delta, q0, F\rangle$, we use LTS(M) to denote the LTS $\langle Q, \alpha M, \delta, q0\rangle$ defined by its first four fields. Note that this transformation does not preserve the language of the FSM. On the other hand, an LTS is in fact a special instance of an FSM, since it can be viewed as an FSM for which all states are accepting. From now on, whenever we apply operators between FSMs and LTSs, it is implied that the LTS is treated as its corresponding FSM.

We call an FSM $M$ *deterministic* iff LTS(M) is deterministic.

**Parallel Composition.** Let $M_1 = \langle Q_1, \alpha M_1, \delta_1, q0_1, F_1\rangle$ and $M_2 = \langle Q_2, \alpha M_2, \delta_2, q0_2, F_2\rangle$ be two FSMs. Then $M_1 \| M_2$ is an FSM $M = \langle Q, \alpha M, \delta, q0, F\rangle$, where:

- $\langle Q, \alpha M, \delta, q0\rangle = LTS(M_1) \| LTS(M_2)$, and
- $F = \{(s_1, s_2) \in Q_1 \times Q_2 \mid s_1 \in F_1 \land s_2 \in F_2\}$.

**Note.** $\mathcal{L}(M_1 \| M_2) = \{t \mid t{\upharpoonright}\alpha M_1 \in \mathcal{L}(M_1) \land t{\upharpoonright}\alpha M_2 \in \mathcal{L}(M_2) \land t \in (\alpha M_1 \cup \alpha M_2)^*\}$

**Satisfiability.** For FSMs $M$ and $P$ where $\alpha P \subseteq \alpha M$, $M \models P$ if and only if $\forall t \in \mathcal{L}(M).t{\upharpoonright}\alpha P \in \mathcal{L}(P)$.

**Complementation.** The complement of an FSM (or an LTS) $M$, denoted $coM$, is an FSM that accepts the complement of $M$'s language. It is constructed by first making

$M$ deterministic, subsequently completing it with respect to $\alpha M$, and finally turning all accepting states into non-accepting ones, and vice-versa. An automaton is complete with respect to some alphabet if every state has an outgoing transition for each action in the alphabet. Completion typically introduces a non-accepting state and appropriate transitions to that state.

## 2.3 The L* Algorithm

In Section 4, we present a framework that automates compositional reasoning using a learning algorithm.

The learning algorithm (L*) used by our approach was developed by Angluin [2] and later improved by Rivest and Schapire [22]. L* learns an unknown regular language ($U$ over an alphabet $\Sigma$) and produces a deterministic FSM $C$ such that $\mathcal{L}(C) = U$. L* works by incrementally producing a sequence of candidate deterministic FSMs $C_1$, $C_2$, ... converging to $C$. In order to learn $U$, L* needs a *Teacher* to answer two type of questions. The first type is a *membership query*, consisting of a string $\sigma \in \Sigma^*$; the answer is *true* if $\sigma \in U$, and *false* otherwise. The second type of question is a *conjecture*, i.e. a candidate deterministic FSM $C$ whose language the algorithm believes to be identical to $U$. The answer is *true* if $\mathcal{L}(C) = U$. Otherwise the Teacher returns a counterexample, which is a string $\sigma$ in the symmetric difference of $\mathcal{L}(C)$ and $U$.

At a higher level, L* creates a table where it incrementally records whether strings in $\Sigma^*$ belong to $U$. It does this by making membership queries to the Teacher. At various stages L* decides to make a conjecture. It constructs a candidate automaton $C$ based on the information contained in the table and asks the Teacher whether the conjecture is correct. If it is, the algorithm terminates. Otherwise, L* uses the counterexample returned by the Teacher to extend the table with strings that witness differences between $\mathcal{L}(C)$ and $U$.

L* is guaranteed to terminate with a minimal automaton $C$ for the unknown language $U$. Moreover, each candidate deterministic FSM $C_i$ that L* constructs is smallest, in the sense that any other deterministic FSM consistent with the table has at least as many states as $C_i$. The candidates conjectured by L* strictly increase in size; each candidate is smaller than the next one, and all incorrect candidates are smaller than $C$. Therefore, if $C$ has $n$ states, L* makes at most $n-1$ incorrect conjectures.

# 3. COMPOSITIONAL PROOF RULES

## 3.1 Motivation

In our previous work on assumption generation and learning [12, 8], we used the following basic rule for establishing that a property $P$ holds for a (closed) parallel composition of two software components $M_1$ and $M_2$.

*Rule 0.*

$$\begin{array}{ll} 1: & M_1 \parallel A_{M_1} \models P \\ 2: & M_2 \models A_{M_1} \\ \hline & M_1 \parallel M_2 \models P \end{array}$$

$A_{M_1}$ denotes an assumption about the environment in which $M_1$ is placed.

In [12], we present an approach to synthesizing the assumption that a component needs to make about its environment for a given property to be satisfied. The assumption produced is the *weakest*, that is, it restricts the environment no more and no less than is necessary for the component to satisfy the property. The automatic generation of weakest assumptions has direct application to the assume-guarantee proof. More specifically, it removes the burden of specifying assumptions manually thus automating this type of reasoning.

The algorithm presented in [12] does not compute partial results, meaning no assumption is obtained if the computation runs out of memory, which may happen if the state-space of the component is too large. We address this problem in [8], where we present a novel framework for performing assume-guarantee reasoning using the above rule in an incremental and fully automatic fashion. The framework iterates a process based on gradually *learning* assumptions. The learning process is based on queries to component $M_1$ and on counterexamples obtained by model checking $M_1$ and its environment, i.e. component $M_2$, alternately. Each iteration may conclude that the required property is satisfied or violated in the system analyzed. This process is guaranteed to terminate; in fact, it converges to an assumption that is necessary and sufficient for the property to hold in the specific system.

Although sound and complete, Rule 0 is unsatisfactory from an automation point of view [2] since it is not symmetric. We thus considered whether some form of "circular", assume-guarantee like, rule could be developed. For our framework the obvious rule for the parallel composition of two processes, where the assumption of each process is discharged by the commitment (or guarantee) of the other, however, is unsound. Indeed, we demonstrate the unsoundness of the following rule.

*Rule 0m.*

$$\begin{array}{ll} 1: & M_1 \parallel A_{M_1} \models P \\ 2: & M_2 \parallel A_{M_2} \models P \\ 3: & P \models A_{M_1} \\ 4: & P \models A_{M_2} \\ \hline & M_1 \parallel M_2 \models P \end{array}$$

Take $M_1$ and $M_2$ each to be the same process $M$ and the property $P$ as illustrated in Figure 1.

Now take as assumption $A_{M_1}$ the behaviour defined by $P$, similarly for $A_{M_2}$. Clearly, premises 3 and 4 hold. And premises 1 and 2 also hold; the parallel composition of $M_1$ with the assumption $A_{M_1}$ constrains its behaviour to be just that of $P$, similarly for premise 2. But unfortunately the conclusion doesn't hold since, in our framework, $M_1$ composed in parallel with $M_2$ is the behaviour $M$ again; $M$ clearly violates property $P$ since it allows $b$ to occur

[2]It is also unsatisfactory from a formal development point of view!

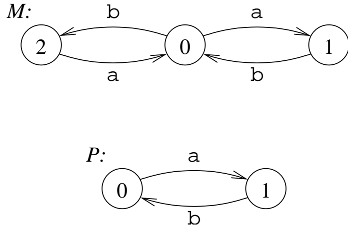



**Figure 1: Example of process $M$ and property $P$ to demonstrate unsoundness of *Rule 0m***

first, rather than ensuring $a$ does. The circular reasoning to discharge the assumptions in this case was unsound. The above rule fails for our framework essentially because the two components may have common erroneous behaviour(as far as the property is concerned) which is (mis-)ruled out by assumptions that are overly presumptuous for the particular composition.

### 3.2 Basic Proof Rule

In the following we give a symmetric parallel composition rule and establish its soundness and completeness for our framework. In Section 4 we then outline how the rule can be used for automated compositional verification along similar lines to the approach given in [8].

*Rule 1.*

$$\begin{array}{ll} 1: & M_1 \parallel A_{M_1} \models P \\ 2: & M_2 \parallel A_{M_2} \models P \\ 3: & \mathcal{L}\,(coA_{M_1} \parallel coA_{M_2}) = \emptyset \\ \hline & M_1 \parallel M_2 \models P \end{array}$$

$M_1$, $M_2$, $A_{M_1}$, $A_{M_2}$ and $P$ are LTSs[3] as defined in the previous section; we require $\alpha P \subseteq \alpha M_1 \cup \alpha M_2$, $\alpha A_{M_1} \subseteq (\alpha M_1 \cap \alpha M_2) \cup \alpha P$ and $\alpha A_{M_2} \subseteq (\alpha M_1 \cap \alpha M_2) \cup \alpha P$. Informally, however, the $A_{M_i}$ are postulated environment assumptions for the components $M_i$ to achieve, respectively, property $P$. $coA_{M_1}$ denotes the co-assumption for $M_1$, which is the complement of $A_{M_1}$. Similarly for $coA_{M_2}$.

The intuition behind premise 3 stems directly from an understanding of the failure of Rule 0m; premise 3 ensures that the assumptions do not both rule out possible, common, violating behaviour from the components. For example, Rule 0m failed in our example above, because both assumptions ruled out common behaviour $(ba)^*$ of $M_1$ and $M_2$, which violates property $P$. Premise 3 in Rule 1 is a remedy for this problem.

THEOREM 1. *Rule 1 is sound and complete.*

PROOF. To establish soundness, we show that the premises together with the negated conclusion leads to a contradiction. Consider a word $t$ for which the conclusion fails, i.e. $t$ is a trace of $M_1 \parallel M_2$ that violates property $P$, in other words $t$ is not accepted by $P$. Clearly, by definition of parallel composition, $t \upharpoonright \alpha M_1$ is accepted by $M_1$. Hence, by premise 1, the trace $t \upharpoonright \alpha A_{M_1}$ can not be accepted by $A_{M_1}$, i.e. $t \upharpoonright \alpha A_{M_1}$ is accepted by $coA_{M_1}$. Similarly, by premise 2, the trace $t \upharpoonright \alpha A_{M_2}$ is accepted by $coA_{M_2}$. By the definition of parallel composition and the fact that an FSM and its complement have the same alphabet, $t \upharpoonright (\alpha A_{M_1} \cup A_{M_2})$ will be accepted by $coA_{M_1} \parallel coA_{M_2}$. But premise 3 states that there are no common words in the co-sets. Hence we have a contradiction.

Our argument for the completeness of Rule 1 relies upon the use of weakest environment assumptions that are constructed in a similar way to [12]. Let $WA(M,P)$ denote the weakest environment for $M$ that will achieve property $P$. $WA(M,P)$ is such that, for any environment $A$, $M \parallel A \models P$ iff $A \models WA(M,P)$.

LEMMA 1. *$coWA(M,P)$ is the set of all traces over the alphabet of $WA(M,P)$ in the context of which $M$ violates property $P$. In other words, this defines the most general violating environment for $(M,P)$. A violating environment for $(M,P)$ is one that causes $M$ to violate property $P$ in all circumstances.*

To establish completeness, we assume the conclusion of the rule and show that we can construct assumptions that will satisfy the premises of the rule. In fact, we construct the weakest assumptions $WA_{M_1}$[4], resp. $WA_{M_2}$, for $M_1$, resp. $M_2$, to achieve $P$, and substitute them for $A_{M_1}$ and $A_{M_2}$. Clearly premises 1 and 2 are satisfied. It remains to show that premise 3 holds. Again we proceed by proof by contradiction. Suppose there is a word $t$ in $\mathcal{L}\,(coWA_{M_1} \parallel coWA_{M_2})$. By definition of parallel composition, t is accepted by both $coWA_{M_1}$ and $coWA_{M_2}$. By Lemma 1, $t \upharpoonright \alpha P$ violates property $P$. Furthermore, there will exist $t_1 \in \mathcal{L}\,(M_1 \parallel coP)$ such that $t_1 \upharpoonright \alpha t = t$, where $\alpha t$ is the alphabet of the assumptions. Similarly for $t_2 \in \mathcal{L}\,(M_2 \parallel coP)$. $t_1$ and $t_2$ can then be combined to be a trace $t_3$ of $M_1 \parallel M_2$ such that $t_3 \upharpoonright \alpha t = t$. But if that is so, this contradicts the assumed conclusion that $M_1 \parallel M_2 \models P$, since $t$ violates $P$. Therefore, there can not be such a common word $t$ and premise 3 holds. □

## 4. AUTOMATED REASONING

### 4.1 Framework

For the use of *Rule 1* to be justified, the assumptions $A_{M_1}$ and $A_{M_2}$ must be more abstract than the components that they represent, i.e. $M_2$ and $M_1$ respectively, but also strong enough for the three steps of the rule to be satisfied. Developing such assumptions is a non-trivial process. We propose an iterative approach to automate the application of Rule 1. The approach extends the framework of counterexample-based learning presented in [8]. As in our previous work and as supported by the LTSA model checking tool [19], we assume that both properties and assumptions are described by deterministic FSMs; this is not a serious limitation since any non-deterministic FSM can be transformed to a deterministic one via the subset construction.

[3]except for when $A_{M_1}$, $A_{M_2}$ and $P$ are false, in which case they are represented as FSMs

[4]Since the context is clear we abbreviate $WA(M,P)$ as $WA_M$.

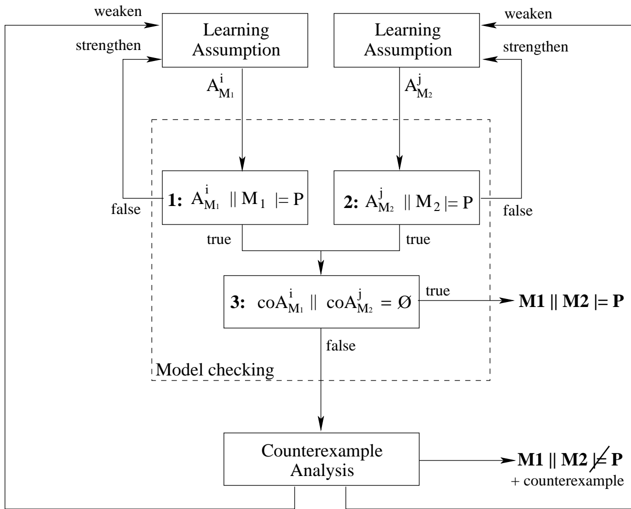



**Figure 2: Incremental compositional verification**

To obtain appropriate assumptions, our framework applies the compositional rule in an iterative fashion as illustrated in Fig. 2. We use a learning algorithm to generate incrementally an assumption for each component, each of which is strong enough to establish the property $P$, i.e. to discharge premises 1 and 2 of Rule 1.

We have seen in the previous section that Rule 1 is guaranteed to return conclusive results with the *weakest assumptions* $WA_{M_1}$, resp. $WA_{M_2}$, for $M_1$, resp. $M_2$, to achieve $P$. We therefore use L* to iteratively learn the traces of $WA_{M_1}$, resp. $WA_{M_2}$. Conjectures are intermediate assumptions $A^i_{M_1}$, resp. $A^j_{M_2}$. As in [8], we use model checking to implement the Teacher needed by L*.

At each iteration, L* is used to build approximate assumptions $A^i_{M_1}$ and $A^j_{M_2}$, based on *querying* the system and on the results of the previous iteration. The first two premises of the compositional rule are then checked. Premise 1 is checked to determine whether $M_1$ guarantees $P$ in environments that satisfy $A^i_{M_1}$. If the result is false, it means that this assumption is too *weak*, i.e. $A^i_{M_1}$ does not restrict the environment enough for $P$ to be satisfied. The assumption therefore needs to be *strengthened*, which corresponds to removing behaviours from it, with the help of the counterexample produced by checking premise 1. In the context of the next assumption $A^{i+1}_{M_1}$, component $M_1$ should at least not exhibit the violating behaviour reflected by this counterexample. Premise 2 is checked in a similar fashion, to obtain an assumption $A^j_{M_2}$ such that component $M_2$ guarantees $P$ in environments that satisfy $A^j_{M_2}$.

If both premise 1 and premise 2 hold, it means that $A^i_{M_1}$ and $A^j_{M_2}$ are strong enough for the property to be satisfied. To complete the proof, premise 3 must be discharged. If premise 3 holds, then the compositional rule guarantees that $P$ holds in $M_1 \parallel M_2$. If it doesn't hold, further analysis is required to identify whether $P$ is indeed violated in $M_1 \parallel M_2$ or whether either $A^i_{M_1}$ or $A^j_{M_2}$ are stronger than necessary. Such analysis is based on the counterexample returned by checking premise 3 and is described in more detail below. If an assumption is too strong it must be *weakened*, i.e. behaviors must be added, in the next iteration. The result of such weakening will be that at least the behavior that the counterexample represents will be allowed by the respective assumption produced at the next iteration. The new assumption may of course be too weak, and therefore the entire process must be repeated.

## 4.2 Counterexample analysis

If premise 3 fails, then we can obtain a counterexample in the form of a trace $t$. Similar to [8], we analyse the trace in order to determine how to proceed. We need to determine whether the trace $t$ indeed corresponds to a violation in $M_1||M_2$. This is checked by simulating $t$ on $M_i||coP$, for $i = 1, 2$. The following cases arise. *(1)* If $t$ is a violating trace of both $M_1$ and $M_2$, then $M_1$ and $M_2$ do indeed have a common bad trace and therefore do not compose to achieve $P$. *(2)* If $t$ is not a violating trace of $M_1$ or $M_2$ then we use $t$ to weaken the corresponding assumption(s).

## 4.3 Discussion

A characteristic of L* that makes it particularly attractive for our framework is its monotonicity. This means that the intermediate candidate assumptions that are generated increase in size; each assumption is smaller than the next one, i.e. $|A^i_{M_1}| \leq |A^{i+1}_{M_1}| \leq |WA_{M_1}|$ and $|A^j_{M_2}| \leq |A^{j+1}_{M_2}| \leq |WA_{M_2}|$. However, we should note that there is no monotonicity at the semantic level, i.e. it is not necessarily the case that $\mathcal{L}(A^i_{M_1}) \subseteq \mathcal{L}(A^{i+1}_{M_1})$ or $\mathcal{L}(A^j_{M_2}) \subseteq \mathcal{L}(A^{j+1}_{M_2})$ hold.

The iterative process performed by our framework terminates for the following reason. At any iteration, our algorithm returns true or false and terminates, or continues by providing a counterexample to L*. By the correctness of L*, we are guaranteed that if it keeps receiving counterexamples, it will eventually, produce $WA_{M_1}$ and $WA_{M_2}$ respectively.

During this last iteration, premises 1 and 2 will hold by definition of the weakest assumptions. The Teacher will therefore check premise 3, which will return either true and terminate, or a counterexample. Since the weakest assumptions are used, by the completeness proof of *Rule 1*, we know that the counterexample analysis will reveal a true error, and hence the process will terminate.

It is interesting to note that our algorithm may terminate before the weakest assumptions are constructed via the iterative learning and refinement process. It terminates as soon as two assumptions have been constructed that are strong enough to discharge the first two premises but weak enough for the third premise to produce conclusive results, i.e. to prove the property or produce a real counterexample; these assumptions are smaller (in size) than the weakest assumptions.

# 5. VARIATIONS

In Section 3 we established that Rule 1 is sound and complete for our framework and in Section 4 we showed its applicability for the automated learning approach to compositional verification. However, we need to explore and understand its effectiveness in our automated compositional verifi-

cation approach. In this section we introduce some straightforward modifications to the rule, maintaining soundness and completeness of course, that may remove unnecessary assumption refinement steps and therefore result in a probable overall improvement in performance.

## 5.1 First Modification

Our first variation, Rule 1a given below, relaxes the third premise by requiring that any common "bad" trace, as far as the assumptions are concerned, satisfies the property $P$. The intuition behind this is that the assumptions may well have been overly restrictive and therefore there may be common behaviours of $M_1$ and $M_2$, ruled out by the assumptions, that do indeed satisfy the property $P$.

*Rule 1a.*

$$\begin{array}{rl} 1: & M_1 \parallel A_{M_1} \models P \\ 2: & M_2 \parallel A_{M_2} \models P \\ 3: & \mathcal{L}(coA_{M_1} \parallel coA_{M_2}) \subseteq \mathcal{L}(P) \\ \hline & M_1 \parallel M_2 \models P \end{array}$$

THEOREM 2. *Rule 1a is sound and complete.*

PROOF. Follows easily from the soundness and completeness proofs for Rule 1. □

*Rule 1b.*

$$\begin{array}{rl} 1: & M_1 \parallel A_{M_1} \models P \\ 2: & M_2 \parallel A_{M_2} \models P \\ 3: & M_1 \parallel coA_{M_1} \models A_{M_2} \textbf{ or } M_2 \parallel coA_{M_2} \models A_{M_1} \\ \hline & M_1 \parallel M_2 \models P \end{array}$$

In essence, in this variation, premise 3 effectively now checks whether any trace in the intersection of the co-assumptions is an illegal behaviour of either component, rather than it just satisfying the property. Notice that the disjunct $M_1 \parallel coA_{M_1} \models A_{M_2}$ is equivalent to $\mathcal{L}(coA_{M_1} \parallel coA_{M_2}) \subseteq \overline{\mathcal{L}(M_1)}$, similarly for the other disjunct. We've used this particular form for the disjuncts because of similarity with assumption discharge.

THEOREM 3. *Rule 1b is sound and complete.*

PROOF. Similar to proofs of Theorems 1 and 2. □

### *Incorporation of Rules 1a and 1b.*

Rule 1a can easily be incorporated into our incremental compositional verification framework. Step 3 of Fig. 2 is followed by an extra step, Step 4, for the case when the intersection of the co-assumptions is not empty. Step 4 checks whether the intersection satisfies the given property: if it returns true then we terminate, otherwise continue with counter-example analysis and assumption refinement. In order to incorporate Rule 1b, we simply include a further check to discharge one of the disjuncts of the rule's third premise.

Clearly these "optimisation"s may result in the verification process terminating after fewer learning iterations. On the other hand there will be some increased overhead in performing the extra checks on each weakening iteration. These issues will be analysed more fully in our future implementation of this incremental approach.

## 5.2 Further Variation

Suppose we are now given components, $M_1$ and $M_2$, with associated properties, $P_1$ and $P_2$. The following composition rule can be used to establish that property$P_1 \parallel P_2$ holds for $M_1 \parallel M_2$.

*Rule 2.*

$$\begin{array}{rl} 1: & M_1 \parallel A_{M_1} \models P_1 \\ 2: & M_2 \parallel A_{M_2} \models P_2 \\ 3: & M_1 \parallel A_{M_1} \models A_{M_2} \\ 4: & M_2 \parallel A_{M_2} \models A_{M_1} \\ 5: & \mathcal{L}(coA_{M_1} \parallel coA_{M_2}) = \emptyset \\ \hline & M_1 \parallel M_2 \models P_1 \parallel P_2 \end{array}$$

where we require $\alpha P_1 \subseteq \alpha M_1$, $\alpha P_2 \subseteq \alpha M_2$, $\alpha A_{M_1} \subseteq \alpha M_1 \cap \alpha M_2$ and $\alpha A_{M_2} \subseteq \alpha M_1 \cap \alpha M_2$.

THEOREM 4. *Rule 2 is sound and complete.*

PROOF. Soundness is established by contradiction, in a similar way to the soundness results for Rules 1, 1a and 1b. We outline the steps. We also abuse and simplify notation by omitting the projections of traces onto the appropriate alphabets.

We assume the properties $P_1$ and $P_2$ are not contradictory, i.e. $\mathcal{L}(P_1 \parallel P_2)$ is not empty, or all behaviours are not erroneous. Further, assume the conclusion does not hold, i.e. $M_1 \parallel M_2 \not\models P_1 \parallel P_2$. There then exists a trace $t$ of $M_1 \parallel M_2$ s.t. $t$ is in not accepted by $P_1 \parallel P_2$. There are three subcases to consider.

1. $t$ not in $P_1$ and $t$ not in $P_2$
2. $t$ not in $P_1$ and $t$ in $P_2$
3. $t$ in $P_1$ and $t$ not in $P_2$

The first case contradicts premise 5. By premise 1, $t$ not in $P_1$ means $t$ is not a trace of $M_1 \parallel A_{M_1}$. But since $t$ is a trace of $M_1 \parallel M_2$ and hence of $M_1$, then $t$ must be accepted by $coA_{M_1}$. Similarly, by premise 2, $t$ must be accepted by $coA_{M_2}$. But this now contradicts premise 5.

For the second case, and similarly for the third case, we will show a contradiction of premise 4, resp. premise 3. As for the first case, by premise 1 if $t$ is not in $P_1$ and $t$ in $M_1$ then $t$ must be accepted by $coA_{M_1}$. As $t$ in $P_2$, $t$ is accepted by $M_2 \parallel A_{M_2}$. Hence, by premise 4, $t$ is in $A_{M_1}$. But $t$ can't be both in $A_{M_1}$ and in $coA_{M_1}$. The mirror argument follows for the third case.

Observe that if premises 3 and 4 were not present, as in the case of rule 1, then soundness is not obtained.

Completeness follows by constructing the weakest assumptions $WA_{M_1}$, resp. $WA_{M_2}$, for $M_1$, resp. $M_2$, to achieve $P_1$, resp. $P_2$, and substituting them for $A_{M_1}$ and $A_{M_2}$. We can then show that if the rule's conclusion holds, then so do the premises. □

It is interesting to note that if premises 3 and 4 of Rule 2 are modified to be in the more usual form of guarantee discharging assumption, i.e. $P_1 \models A_{M_2}$ and $P_2 \models A_{M_1}$, then the rule is not complete.

As was the case with Rule 1, we can weaken premise 5 of Rule 2 to obtain similar rules to Rule 1a and Rule 1b.

# 6. HISTORICAL PERSPECTIVE

Over two decades ago, the quest for obtaining sound and complete compositional program proof systems, in various frameworks, remained open. The foundational work on proof systems for concurrent programs, for example [3, 20, 18], whilst not achieving compositional rules, introduced key notions of meta-level co-operation proofs and non-interference proofs. These meta-level proofs were carried out using program code and intermediate assertions from the proofs of the sequential processes. Assumption-commitment, or rely-guarantee, style specifications, in addition to pre- and post-conditions, were then introduced to capture the essence of the meta-level co-operation and non-interference proofs, lifting the assumptions that were implicitly made in the sequential proof outlines to be an explicit part of the specification. Program proof systems, built over such extended specifications, were then developed to support the stepwise, or hierarchical, development of concurrent, or distributed, programs, see for example [16, 25, 4, 23]. The development of such compositional proof systems continues to this day and the interested reader should consult [10] for an extensive and detailed coverage.

In recent years, there has been a resurgence of interest in formal techniques, and in particular assume-guarantee reasoning, for supporting component-based design: see for example [9]. Even though various sound and often complete proof systems have been developed for this style of reasoning, more often than not it is a mental challenge to obtain the most appropriate assumptions [15]. It is even more of a challenge to find automated techniques to support this style of reasoning. The thread modular reasoning underlying the Calvin tool [11] is one start in this direction. One way of addressing both the design and verification of large systems is to use their natural decomposition into components. Formal techniques for support of component-based design are gaining prominence, see for example [9]. In order to reason formally about components in isolation, some form of assumption (either implicit or explicit) about the interaction with, or interference from, the environment has to be made. Even though we have sound and complete reasoning systems for assume-guarantee reasoning, see for example [16, 21, 6, 14], it is always a mental challenge to obtain the most appropriate assumption [15].

It is even more of a challenge to find automated techniques to support this style of reasoning. The thread modular reasoning underlying the Calvin tool [11] is one start in this direction. The Mocha toolkit [1] provides support for modular verification of components.

The problem of generating an assumption for a component is similar to the problem of generating component interfaces to deal with intermediate state explosion in CRA. Several approaches have been defined for automatically abstracting a component's environment to obtain interfaces [5, 17]. These approaches do not address the incremental refinement of interfaces.

Learning in the context of model checking has also been investigated in [13], but with a different goal. In that work, the L* Algorithm is used to generate a model of a software system which can then be fed to a model checker. A conformance checker determines if the model accurately describes the system.

# 7. CONCLUSIONS AND FUTURE WORK

Although theoretical frameworks for sound and complete assumption-commitment reasoning have existed for many years, their practical impact has been limited because they involve non-trivial human interaction. In this paper, we have presented a new set of sound and complete proof rules for parallel composition that support a fully automated verification approach based upon such a reasoning style. The automation approach extends and improves upon our previous work that introduced a learning algorithm to generate and refine assumptions based on queries and counterexamples, in an iterative process. The process is guaranteed to terminate, and return true if a property holds in a system, and a counterexample otherwise. If memory is insufficient to reach termination, intermediate assumptions are generated, which may be useful in approximating the requirements that a component places on its environment to satisfy certain properties.

One advantage of our approach is its generality. It relies on standard features of model checkers, and could therefore easily be introduced in any such tool. For example, we are currently in the process of implementing it in the LTSA. The architecture of our framework is modular, so its components can easily be substituted by more efficient ones.

We have implemented our framework within the LTSA tool and over the coming months we will conduct a number of experiments to establish the practical effectiveness of our new composition rule and its variations. We need to understand better the various trade-offs between the increased overhead of additional premise testing and the computational savings from earlier termination of the overall process. In addition, we need to investigate known variants of our rules for $N$-process compositions, again considering various practical tradeoffs in implementation terms. Of course, an interesting challenge will also be to extend the types of properties that our framework can handle to include liveness, fairness, and timed properties.

# REFERENCES

[1] R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. MOCHA: Modularity in model checking. In *Proc. of the Tenth Int. Conf. on*

*Comp.-Aided Verification (CAV)*, pages 521–525, June 28–July 2, 1998.

[2] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, Nov. 1987.

[3] K. R. Apt, N. Francez, and W.-P. de Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 2:359–385, 1980.

[4] H. Barringer and R. Kuiper. Hierarchical development of concurrent systems in a framework. In S. B. et al, editor, *Seminar in Concurrency*, volume 197 of *Lecture Notes in Computer Science*, pages 35–61, 1985.

[5] S. C. Cheung and J. Kramer. Context constraints for compositional reachability analysis. *ACM Trans. on Soft. Eng. and Methodology*, 5(4):334–377, Oct. 1996.

[6] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Proc. of the Fourth Symp. on Logic in Comp. Sci.*, pages 353–362, June 1989.

[7] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[8] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning assumptions for compositional verification. In *9th International Conference for the Construction and Analysis of Systems (TACAS 2003)*, volume 2619 of *LNCS*, Warsaw, Poland, 2003. Springer.

[9] L. de Alfaro and T. A. Henzinger. Interface theories for component-based design. In *Proc. of the First Int. Workshop on Embedded Soft.*, pages 148–165, Oct. 2001.

[10] W.-P. de Roever, F. de Boer, U. Hanneman, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Non-compositional Methods*. Cambridge University Press, 2001.

[11] C. Flanagan, S. N. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. In *Proc. of the Eleventh European Symp. on Prog.*, pages 262–277, Apr. 2002.

[12] D. Giannakopoulou, C. S. Păsăreanu, and H. Barringer. Assumption generation for software component verification. In *Proc. of the Seventeenth IEEE Int. Conf. on Auto. Soft. Eng.*, Sept. 2002.

[13] A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. In *Proc. of the Eighth Int. Conf. on Tools and Alg. for the Construction and Analysis of Sys.*, pages 357–370, Apr. 2002.

[14] O. Grumberg and D. E. Long. Model checking and modular verification. In *Proc. of the Second Int. Conf. on Concurrency Theory*, pages 250–265, Aug. 1991.

[15] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. You assume, we guarantee: Methodology and case studies. In *Proc. of the Tenth Int. Conf. on Comp.-Aided Verification (CAV)*, pages 440–451, June 28–July 2, 1998.

[16] C. B. Jones. Specification and design of (parallel) programs. In R. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Congress*, pages 321–332. IFIP: North Holland, 1983.

[17] J.-P. Krimm and L. Mounier. Compositional state space generation from Lotos programs. In *Proc. of the Third Int. Workshop on Tools and Alg. for the Construction and Analysis of Sys.*, pages 239–258, Apr. 1997.

[18] G. Levin and D. Gries. A proof technique for communicating sequential processe s. *Acta Informatica*, 15(3):281–302, 1981.

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

[20] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.

[21] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. Apt, editor, *Logic and Models of Concurrent Systems*, volume 13, pages 123–144, New York, 1984. Springer-Verlag.

[22] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, Apr. 1993.

[23] E. W. Stark. A proof technique for rely/guarantee properties. In *Fifth Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 206 of *Lecture Notes in Theoretical Computer Science*, pages 369–391. Springer-Verlag, Dec. 1985.

[24] Q. Xu, W. P. de Roever, and J. He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.

[25] J. Zwiers, W.-P. de Roever, and P. van Emde Boas. Compositionality and concurrent networks: Soundness and completeness of a proof system. In *Proceedings of ICALP '85, Springer LNCS 194*, pages 509–519. Springer-Verlag, 1985.

# Behavioral Substitutability in Component Frameworks: a Formal Approach

Sabine Moisan
INRIA Sophia Antipolis
2004, route des Lucioles
06902 Sophia Antipolis, France
Sabine.Moisan@inria.fr

Annie Ressouche
INRIA Sophia Antipolis
2004, route des Lucioles
06902 Sophia Antipolis, France
Annie.Ressouche@inria.fr

Jean-Paul Rigault
I3S Laboratory
UNSA/CNRS, UMR 6070
06902 Sophia Antipolis, France
jpr@essi.fr

## ABSTRACT

When using a component framework, developers need to respect the behavior implemented by the components. Dynamic information such as the description of valid sequences of operations is required. In this paper we propose a mathematical model and a formal language to describe the knowledge about behavior. We rely on a hierarchical model of deterministic finite state-machines. The communication between the machines follows the Synchronous Paradigm. We focus on extension of components, owing to the notion of behavioral substitutability. Our approach relies on compositionality properties to ease automatic verification. From the language and the model, we can draw practical design rules that preserve safety properties. Associated tools may ensure correct and safe reuse of components, as well as automatic simulation and verification, code generation, and run-time checks.

## Keywords

framework, components, behavioral substitutability, synchronous reactive systems, model checking

## 1. INTRODUCTION

A current trend in Software Engineering is to favor reusability of code and also of analysis and design models. This is mandatory to improve product time to market, software quality, maintenance, and to decrease development cost. The notion of frameworks was introduced as a possible answer to these needs. Basically, a framework is a well-defined architecture composed of generic classes and their relationships. As reusable entities, classes rapidly appeared as too fine grained. Hence, the notion of component frameworks emerged. According to Szyperski [21] a component is "a unit of [software] composition with contractually specified interfaces and explicit context dependencies...". In the object-oriented approach a component usually corresponds to a collection of interrelated classes and objects providing a logically consistent set of services.

Using a component framework involves selecting, adapting, and assembling components to build a customized application. Thus *reusing* existing components is a major task. Building on reusability is not straightforward, though; it implies to understand the nature of the contract between the client (i.e., the framework user) and the component. This contract may be the mere specification of a static interface (list of operation signatures), which is clearly not sufficient since it misses the information regarding the component *behavior*. Adding pre- and post-conditions to operations is an interesting improvement. However, contracts express only behavior local to an operation, making it difficult to comprehend the global valid sequences of operations. The description of such a valid sequence is the essential part of what we call the *protocol of use* of the framework. This protocol is often more complex than for using, e.g., a simple library. Hence, it is important to provide models and tools to formalize it, reason about it, and manipulate it.

Our work on formalizing component protocols relies on our experience with a framework for knowledge-based system (KBS) inference engines, named BLOCKS [17]. BLOCKS's objective is to help designers create new engines and reuse or modify existing ones, without extensive code rewriting. It is a set of C++ classes, each one coming with a behavioral description of the valid sequences of operations, in the form of state-transition diagrams. Such a description allowed us to prove invariant properties of the framework, using model-checking techniques. As with other frameworks, the developer adapts BLOCKS classes essentially through subtyping (more exactly, class derivation used as subtyping). The least that can be expected is that the derived classes respect the behavioral protocol that the base classes implement and guarantee. In particular, we want to ensure that an invariant property at the framework base level also holds at the developer's class level. Thus the notion of *behavioral substitutability* is central to such a safe use of the framework. To this end we chose to elaborate a formal model of behavioral substitutability so that we may lay design rules on top of it. In this model safety properties are preserved during subtyping. Our aim is to propose a verification algorithm as well as practical design rules to ensure sound framework adaptation.

The paper is organized as follows. The next section details our notion of components and defines their protocols of use. Section 3 presents the mathematical model and the formal language to describe the behavioral part of the protocol. Section 4 illustrates practical design rules drawn from the model. Section 5 briefly compares our approach with other techniques and methods. The conclusion draws perspectives about the development of supporting tools.

# 2. TARGET FRAMEWORK CHARACTERISTICS

## 2.1 Notion of Components

In the object-oriented community a component framework is usually composed of hierarchies of classes that the framework user may compose or extend. The root class of each hierarchy corresponds to an important concept in the target domain. In this context, a component can be viewed as the realization of a sub-tree of the class hierarchy: this complies with one of Szyperski's definitions for components [21].

As a matter of example, let us examine the problem of history management in an object-oriented environment. In our framework (BLOCKS) a *history* is composed of several successive *snapshots*, each one gathering the modifications (or *deltas*) to object attributes that have happened since the previous snapshot (that is during an execution step). It is a rather general view of history management and any framework with a similar purpose is likely to provide classes such as `History`, `Snapshot` and `Delta`, as shown in the UML class diagram of figure 1. Class `Snapshot` memorizes the modifications of objects during an execution step in its attached `Delta` set; it displays several operations: memorize the deltas and other contextual information, add a new delta, and add a child snapshot (i.e., close the current step and start a new one).

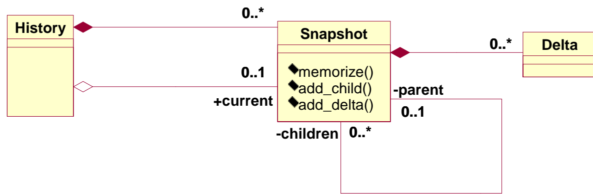



**Figure 1: Simplified UML diagram of class `Snapshot`**

## 2.2 Using a Framework

Framework users both adapt the components and write some glue code.They will (non-exclusively) use these components directly, or specialize the classes they contain by inheritance, or compose the classes together, or instantiate new classes from predefined generic[1] ones. Among all these possibilities, class derivation is frequent. It is also the one that may raise the trickiest problems, that is why we shall concentrate on it in the rest of this paper.

When deriving a class the user may either introduce new attributes and/or operations or redefine inherited operations. These specializations should be "semantically acceptable", i.e., they should respect the framework invariants.

In our example, the `Snapshot` class originally does not take into account a possible "backtrack" (the "linear" history of `Snapshot` becomes a "branching" one). This feature is necessary in simulation activities to try different actions or to modify some contextual information and see what happens. To cope with such requirements, the user can derive a `BSnapshot` class from `Snapshot` (figure 2). In this example, the inherited operations need no redefinition[2]. `BSnapshot` defines two new operations: `regenerate` that reestablishes the memorized values and `search` that checks whether a condition was true in a previous state. The regeneration feature implies that deltas have the ability to redo and undo their changes; hence the new class `BDelta` has to be substituted to `Delta`. Relying on static information in the class diagram of `Snapshot` (signatures of operations and associations), the framework user obtains the inheritance graph shown on figure 2.

[1] class templates in C++

[2] In the general case, there would be new operations as well as redefined operations. Our approach is able to cope with both cases.

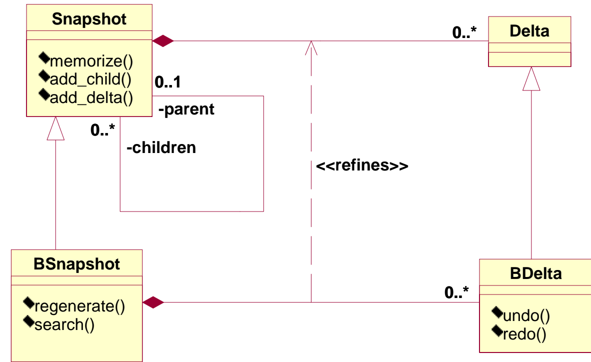



**Figure 2: UML class diagram of `BSnapshot`; above, the original classes, below, the derived ones**

## 2.3 Protocol(s) of Use

Static information is not sufficient to ensure a safe and correct use of a framework: specifying a *protocol of use* is required. This protocol is defined by two sets of constraints. First, a *static* set enforces the internal consistency of class structures. UML-like class diagrams provide a part of this information: input interfaces of classes (list of operation signatures), specializations, associations, indication of operation redefinitions, and even constraints on the operations that a component expects from other components (a sort of *required interface*, something that will likely find its way into UML 2.0). We do not focus on this part of the protocol since its static nature makes it easy to generate the necessary information at compile-time. A second set of constraints describes *dynamic* requirements: (1) the legal sequences of operation calls, (2) the specification of internal behavior of operations and of sequences of messages these operations send to other components, and (3) the behavioral specification of valid redefinition of operations in derived classes. These dynamic aspects are more complicated to express than static ones and there is no tool (as natural as compiler-like tools for the static case) to handle and check them. While item (1) and partially item (2) are addressed by classical UML state-transition models, the complete treatment of the last two items is more challenging. We shall propose a solution in section 3.

# 3. BEHAVIOR DESCRIPTION AND BEHAVIOR REFINEMENT

Our approach is threefold. First, we define a mathematical model providing consistent description of *behavioral entities*. In the model, behavioral entities are whole components, sub-components, single operations, or any assembly of these. Hence, the whole system is a hierarchical composition of communicating behavioral entities. Such a model complements the UML approach and allows to specify the class and operation behavior with respect to class derivation. Second, we propose a *hierarchical* behavioral specification *language* to describe the dynamic aspect of components. In the third place, we define a *semantic* mapping to bridge the gap

between the specification language and its meaning in the mathematical model.

As already mentioned, our primary intent is to formalize the behavior side of class derivation, in the sense of *subtyping*[3]. In the object-oriented approach, subtyping usually obeys the classical Substitutability Principle [13]. This principle has a static interpretation which leads to, for instance, the well-known covariant and contravariant issues for parameters and return types. But it may also be given a dynamic interpretation, leading to behavioral subtyping, or *behavioral substitutability* [10]. This is the kind of interpretation we need to enforce the dynamic aspect of framework protocols, since it provides a notion of behaviorwise safe derivation.

To deal with behavioral substitutability, we need behavior representation formalisms: we propose to rely on the family of *synchronous* models [9]. These models are dedicated to specify event-driven and discrete time systems. Such systems interact with their environment, reacting to input events by sending output events. Furthermore, they obey the *synchrony hypothesis*: the corresponding reaction is *atomic*; during a reaction, the input events are frozen, all events are considered as *simultaneous*, events are broadcast and available to any part of the system that listens to them. A reaction is also called an *instant*. The succession of instants defines a logical time. The major interest of synchronous models is that their verification exhibits a lower computational complexity than asynchronous ones, which is the main reason for our choice.

## 3.1 Mathematical Model of Behavior

Labeled transition systems are usual mathematical models for synchronous languages. These systems are a special kind of finite deterministic state machines (automata) and we shall denote them LFSM for short. In our model, we use LFSMs to represent the state behavior of *behavioral entities* (classes as well as their operations). Each transition has a *label* representing an elementary execution step of the entity, consisting of a *trigger* (input condition) and an *action* to be executed when the transition is fired. In our case an action corresponds to emitting events, such as calling an operation of some component whereas a trigger corresponds to receiving events such as calling an operation.

A LFSM is a tuple $M = (S, s_0, T, A)$ where $S$ is a finite set of states, $s_0 \in S$ is the initial state, $A$ is the *alphabet* of events from which the set of labels $L$ is built, and $T$ is the transition relation $T \subseteq S \times L \times S$. We introduce the set $I$ of input events $I \subseteq A$ and the set $O \subseteq A$ of output events (or actions).

***Labels.*** $L$, the set of *labels*, has elements of the form $i/o$, where $i$ is the trigger set and $o \subseteq O$ the action or output events set; $i$ has the form $(i^+, i^-)$ where $i^+$, the positive (input event) set of a label (resp. $i^-$, the negative (input event) set), consists of the events tested for their presence (resp. for their absence) in the trigger at a given instant.

A trigger contains the information about all the input events, be they present or absent at a given instant. Obviously, an event cannot be tested for both absence and presence at the same instant. Thus $(i^+, i^-)$ constitutes a partition of $I$. Moreover, as a consequence of the previous definition of an *instant* in the synchronous model, an event cannot be tested for absence while being emitted in the same instant. Hence, the following well-formedness conditions on labels:

$$\begin{cases} i^+ \cap i^- = \emptyset & \text{(trigger consistency)} \\ i^+ \cup i^- = I & \text{(trigger completeness)} \\ i^- \cap o = \emptyset & \text{(synchrony hypothesis)} \end{cases}$$

***Transitions.*** Each transition $s \xrightarrow{l} s'$ has three parts: a source state $s$, a label $l$, and a target state $s'$. There cannot be two transitions leaving the same state and bearing the same trigger. Formally, if there are two transitions from the same state $s$ such that $s \overset{i_1/o_1}{\longrightarrow} s_1$ and $s \overset{i_2/o_2}{\longrightarrow} s_2$, with $s_1 \neq s_2$, then $i_1 \neq i_2$. This rule, together with the label well-formedness conditions, ensure that LFSMs are *deterministic*. This requirement for determinism constitutes one of the foundations of the synchronous approach and is mandatory for all models and proofs that follow.

***Behavioral Substitutability.*** The substitutability principle should apply to the dynamic semantics of a behavioral entity–such as either a whole class or one of its (redefined) operations [10, 18]. If $M$ and $M'$ are LFSMs denoting respectively some behavior in a base class and its redefinition in a derivative, we seek for a relation $M' \preceq M$ stating that "$M'$ safely extends $M$". To comply with inheritance, this relation must be a preorder.

Following the substitutability principle, we say that $M'$ is a correct extension of $M$, iff the alphabet of $M'$ ($A'_M$) is a superset of the alphabet of $M$ ($A_M$) and every sequence of inputs that is valid [4] for $M$ is also valid for $M'$ and produces the same outputs (once restricted to the alphabet of $M$). Thus, the behavior of $M'$ restricted to the alphabet of $M$ is identical to the one of $M$. Formally,

$$M' \preceq M \Leftrightarrow A_M \subseteq A_{M'} \wedge M \, \mathcal{R}_{sim} \, (M' \backslash A_M)$$

where $M' \backslash A_M$ is the *restriction* of $M'$ to the alphabet of $M$ and $\mathcal{R}_{sim}$ the behavioral simulation relation. Both are defined below.

First, we define the restriction ($l \backslash A$) of a label ($l$) over an alphabet ($A$) as follows: let $l = i/o$,

$$l \backslash A = \begin{cases} (i \cap A / (o \cap A) & \text{if } i^+ \subseteq A \\ undef & \text{otherwise} \end{cases}$$

Intuitively, this corresponds to consider as undefined all the transitions bearing a positive trigger not in $A$, and to strip the events not in $A$ from the outputs.

The restriction of $M$ to the alphabet $A$ (generally with $A \subseteq A_M$) is obtained by restricting all the labels of $M$ to $A$, then discarding the resulting undefined transitions. Formally, let $M = (S, s_0, T, A_M)$ be a LFSM, $M \backslash A = (S, s_0, T \backslash A, A_M \cap A)$ where $T \backslash A$ is defined as follows:

$$s \xrightarrow{l'} s' \in T \backslash A \Leftrightarrow \exists \, s \xrightarrow{l} s' \in T \wedge l' = l \backslash A \neq undef$$

Second, we adopt a behavioral simulation relation similar to Milner's classical simulation [16]. Let $M_1$ and $M_2$ be two LFSMs with the same alphabet: $M_1 = (S_{M_1}, s_0^{M_1}, T_{M_1}, A)$ and $M_2 =$

[3]Note that, in this paper, derivation, inheritance, specialization all refer to the *subtyping* interpretation. In particular, we do not consider the other uses or interpretations of inheritance that some programming languages may offer.

[4]A *path* in a LFSM $M$ is a (possibly infinite) sequence of transitions $\pi = s_0 \overset{i_0/o_0}{\longrightarrow} s_1 \overset{i_1/o_1}{\longrightarrow} s_2 ...$ such that $\forall i (s_i, i_i/o_i, s_{i+1}) \in T$. The sequence $i_0/o_0, i_1/o_1 ...$ is called the *trace* associated with the path. When such a path exists, the corresponding trigger sequence $i_0, i_1, ...$ is said to be a *valid* sequence of $M$.

$(S_{M_2}, s_0^{M_2}, T_{M_2}, A)$. A relation $\mathcal{R}_{sim} \subseteq S_{M_1} \times S_{M_2}$ is called a *simulation* iff $(s_0^{M_1}, s_0^{M_2}) \in \mathcal{R}_{sim}$ and

$$\forall (s_1, s_2) \in \mathcal{R}_{sim} :$$
$$s_1 \xrightarrow{l} s_1' \in T_{M_1} \Rightarrow \exists s_2 \xrightarrow{l} s_2' \in T_{M_2} \wedge (s_1', s_2') \in \mathcal{R}_{sim}$$

Simulation is local, since the relation between two states is based only on their successors. As a result, it can be checked in polynomial time and it is widely used as an efficient computable condition for trace-containment. Moreover, the simulation relation can be computed using a symbolic fixed point procedure [11], allowing to tackle large-sized state spaces.

We say that $M'$ *simulates* $M$ iff $M' \preceq M$. Thus, $M'$ simulates $M$ iff there exists a relation binding each state of $M$ to a state of the restriction of $M'$ to the alphabet of $M$. Any valid sequence of $M$ is also a valid sequence of $M'$ and the output traces are identical, once restricted to $A_M$. As a consequence, if $M'$ simulates $M$, $M'$ can be substituted for $M$, for all purposes of $M$.

Milner's simulation relation ($\mathcal{R}_{sim}$) is a preorder and preserves satisfaction of the formulae of a subset of temporal logic, expressive enough for most verification tasks (namely $\forall CTL^*$ [12]). Moreover, this subset has efficient model checking algorithms. Obviously, relation $\preceq$ is also a preorder over LFSMs and any formula that holds for $M$ holds also for $M'$.

The notion of correct extension can be extended to components. We can represent the *protocol of use* of a class $C$ (see section 2.3) by a LFSM $\mathcal{P}(C)$. If $C$ and $C'$ are two classes, $C' \preceq C$ iff (1) $C'$ derives from $C$ (according to footnote [3], this means "is a subtype of"), (2) the protocol of use of $C'$ simulates the one of $C$, that is $\mathcal{P}(C') \preceq \mathcal{P}(C)$. As indicated in 2.3, we assume that the protocol of use of a class describes not only the way the other objects may call the class operations, but also the way the operations of the class invoke operations on (other) objects.

With such a model, the description of behavior matches the class hierarchy. Hence, class and operation refinements are compatible and consistent with the static description: checking dynamic behavior may benefit from the static hierarchical organization.

## 3.2 Behavior Description Language

We need a language that makes it possible to describe complex behavioral entities in a structured way, particularly by means of scoping and composition. Our language is very similar to *Argos* [15]. It offers a graphical notation close to UML StateCharts with some restrictions, but with a different semantics based on the Synchronous Paradigm [9]. The language is easily compiled into LFSMs. Programs written in this language operationally describe behavioral entities; we call them *behavioral programs*. The semantics of this language should be expressible in terms of the mathematical model, permitting an easy translation into LFSMs.

The primitive elements from which programs are constructed are called *flat automata*, since they cannot be decomposed (they contain no application of any operators). They are the direct representation of LFSMs, with the following simplified notation: only positive (i.e., present) events appear in triggers, all other events are considered as absent.

The language is generated by the following grammar (where $A$ is a flat automaton, $s$ a state name and $Y$ a set of events):

$$P ::= A \mid A[P/s] \mid P \| P \mid P < Y >$$

*Parallel composition* ($P \| Q$) is a symmetric operator which behaves as the synchronous product of its operands where labels are unioned. *Hierarchical composition* ($A[P/s]$) corresponds to the possibility for a state in an automaton to be refined by a behavioral (sub) program. This operation is able to express preemption, exceptions, and normal termination of sub-programs. *Scoping* ($P < Y >$) where $P$ is a program and $Y$ a set of *local* events, makes it possible to restrict the scope of some events. Indeed, when refining a state by combining hierarchical and parallel composition, it may be useful to send events from one branch of the parallel composition to the other(s) without these events being globally visible. This operation can be seen as encapsulation: local events that fired a transition must be emitted in their scope; they cannot come from the surrounding environment.

The language offers syntactic means to build programs that reflect the behavior of components. Nevertheless, the soundness of the approach requires a clear definition of the relationship between behavioral programs and their mathematical representation as LFSMs (section 3.1). Let $\mathcal{B}$ denote the set of behavioral programs and $\mathcal{M}$ the set of LFSMs. We define a *semantic* function $\mathcal{S} : \mathcal{B} \longrightarrow \mathcal{M}$ that is stable with respect to the previously defined operators (parallel composition, hierarchical composition, and scoping).

$\mathcal{S}$ is *structurally* defined over the syntax of the language. Because of lack of space, we just give here the flavor of the definition of this semantics. For a more complete description, see [20]. A flat automaton constitutes its own semantics. The semantics of parallel composition $P \| Q$ is the *synchronous product* [9] of the semantics of $P$ and $Q$: each reaction (instant) is considered atomic; within an instant, input and output events are matched by name, providing instantaneous communication. The semantics of hierarchical composition $P[Q/s]$ is basically the one of $P$ where state $s$ has been replaced by the semantics of $Q$ whose transitions have been modified to respect the outgoing transitions (preemptions) of $s$. More specifically, in the absence of preemption, the semantics of $Q$ remains the same; otherwise, the preemptions of $s$ have priority, which may lead to unioning internal and preemption actions. For scoping, the semantics of $P < Y >$ is basically the one of $P$ where transitions triggered by local events that are not emitted are discarded and where the occurrences of local events are removed from the labels of the remaining transitions. Thus all events in $Y$ (be they triggers or actions) are encapsulated within $P < Y >$ and invisible from the outside, as is invisible the internal communication they support.

The following theorem expresses that relation $\preceq$ is a congruence with respect to the language operators. The proof [20] is out of the scope of the paper, and is obtained by explicit construction of the preorder relation.

THEOREM 1. *Let $P$, $Q_1$ and $Q_2$ be behavioral programs such that $\mathcal{S}(Q_1) \preceq \mathcal{S}(Q_2)$ and both $P$, $Q_1$ and $P$, $Q_2$ are outputs disjoint; the following holds:*
$\mathcal{S}(P[Q_1/s]) \preceq \mathcal{S}(P[Q_2/s])$
$\mathcal{S}(P \| Q_1) \preceq \mathcal{S}(P \| Q_2)$
$\mathcal{S}(Q_1 < Y >) \preceq \mathcal{S}(Q_2 < Y >)$

This *compositionality property* is fundamental to our approach. It

gives a modular and incremental way to verify behavioral programs using their natural structure: properties of a whole program can be deduced from properties of its sub-programs. This helps to push back the bounds of state explosion, the major drawback of model checking.

## 3.3 Modular Verification

The compositionality property is very useful, since one can deal with highly complex global behaviors provided that they result from composing elementary behaviors that can be verified, modified, and understood incrementally. In particular it makes it possible to perform modular verification using some temporal logic.

*Temporal logics* are formalisms for describing sequences of transitions between states in a finite state machine model. They are formal languages where assertions related to behavior are easily expressed. The logic we consider ($\forall CTL^*$) [12] is based on first-order logic. This logic, to be efficient when deciding whether a formula is true, does not introduce the existential path quantifier. It offers *temporal operators* that make it possible to express properties holding for a given state, for the next state (operator **X**), eventually for a future state (**F**), for all future states (**G**), or that a property remains true until some condition becomes true (**U**). One can also express that a property holds for all the paths starting in a given state ($\forall$). These operators can be combined with boolean connectors and nested.

The logic may be interpreted over LFSMs. One can express model-checking algorithms and satisfaction of a formula is defined in a natural inductive way. We say that a LFSM $M$ satisfies a state formula $\psi$ ($M \models \psi$) if property $\psi$ is true for the initial state of $M$.

In the same line as Clarke et al. [12], the main result of our approach is the following theorem (the proof [20] is by structural induction on formulae and from the translation of LFSMs into Kripke structures).

THEOREM 2. *Let $P$ and $Q$ two behavioral programs with disjoint output sets and $\psi$ a $\forall CTL^*$ formula:*
*if $\mathcal{S}(P) \models \psi$ then $\mathcal{S}(P[Q/s]) \models \psi$.*
*if $\mathcal{S}(P) \models \psi$ or $\mathcal{S}(Q) \models \psi$ then $\mathcal{S}(P \| Q) \models \psi$.*

This result complements theorem 1; it expresses the compositional stability of proofs with respect to the composition operators. This property provides a hierarchical and incremental means to verify properties and is the key to simplify model checking.

The properties that are preserved by our operators include substitutability. This is an immediate consequence of theorems 1 and 2. For instance, if we have proved that $P_1 \preceq P_2$, then we can infer that $P_1 \| Q \preceq P_2 \| Q$, for any possible $Q$, provided that it is output disjoint with $P_1$ and $P_2$.

# 4. PRACTICAL ISSUES

## 4.1 Design Rules

To guarantee a safe use of the components, we state some practical design rules that we can draw from our model and that can be applied at the behavioral language level. When a behavioral program $P$ (called the base program) is extended by another behavioral program $P'$, respecting these rules ensures that we obtain a new deterministic automaton for which behavioral substitutability holds ($P' \preceq P$). These rules correspond to *sufficient* conditions that save us the trouble of a formal proof for each derived program.

At this time we have identified eight such practical rules. A formal description of these rules can be found in [20]. We briefly list them here: (1) no modification of the base program structure (no deletion nor modification of transitions or states); (2) possibility of adding trigger-disjoint transitions for a given state; possibility of parallel composition with a program with (3) disjoint actions or (4) different initial trigger or (5) with a substitutable program; possibility of hierarchical composition with a program (6) without auto-preemption or (7) with disjoint triggers and actions; (8) no localization of global events.

## 4.2 Application to Components

To illustrate our purpose, let us consider the previously mentioned history mechanism (section 2.1). We present on figure 3(a) the behavioral program for the whole `Snapshot` class. This program specifies the valid sequences of operations that can be applied to `Snapshot` instances. Two states correspond to execution of operations (`memorize` and `add_child`); they are to be refined by behavioral programs describing these operations. Figure 3(b) presents the expected behavioral program for `BSnapshot` which derives from class `Snapshot`. In particular, `BSnapshot` necessitates a new operation, `regenerate`, called when backtracking the history (i.e., when `search` returns `success`). It is clear that the new class sports a behavior significantly different from its base class: it has the extra possibilities to search inside a sleeping snapshot and to call `regenerate` when `success` occurs.

The behavioral program of `BSnapshot` has been obtained from the one of `Snapshot` after applying a combination of our design rules. Obviously no state nor transition have been deleted from `Snapshot` (rule 1). The new transition from `inactive` to `regeneration` bears a completely new trigger (rule 2). The program that refines state `inactive` has no trigger belonging to the preemption trigger set of this state (rule 7). Finally, the local event `success` was not part of the `Snapshot` program (rule 8). Thus, by construction, `BSnapshot` is substitutable for class `Snapshot`; no other verification is necessary to assert that `BSnapshot` $\preceq$ `Snapshot`.

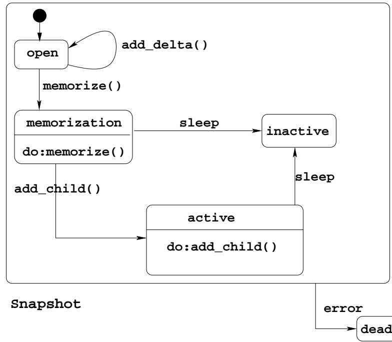
Therefore, even though `BSnapshot` extends `Snapshot` behavior, the extension has no influence when a `BSnapshot` is used as a `Snapshot`. As a result, every trace of `Snapshot` is also a trace of `BSnapshot`.

## 4.3 Stability of Properties

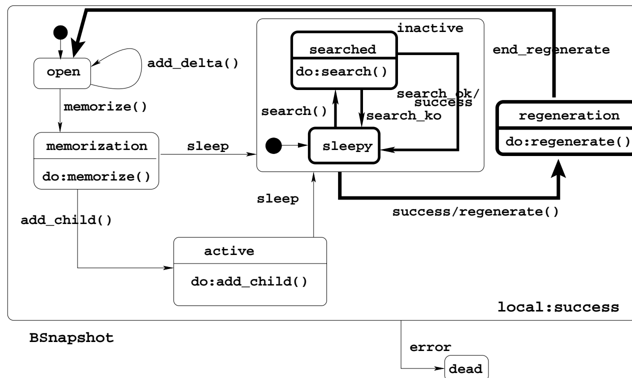
Continuing with the previous example, `BSnapshot` $\preceq$ `Snapshot` implies that every temporal property in $\forall CTL^*$ true for `Snapshot` is also true for its extension `BSnapshot`. For instance, suppose we wish to prove the following property: "It is possible to add a child to a snapshot (i.e., to call the `add_child()` operation) only after memorization has been properly done". Looking at the behavioral program (figure 3(a)), we can decompose $P_{child}$ into two specifications:

$$\forall \mathbf{G}(\texttt{add_child}() \& \forall \mathbf{G}(\neg \texttt{error})) \Rightarrow \forall \mathbf{F} \texttt{state} = \texttt{inactive}$$
$$\forall \mathbf{G}(\texttt{error} \Rightarrow \forall \mathbf{G}(\neg \texttt{state} = \texttt{inactive}))$$

Intuitively, the first formula corresponds to memorization success: if `add_child()` is received and if no `error` occurs, then state `inactive` is reached. The second formula corresponds to memorization failure: `error` occurred, and state `inactive` will never be reached. A model-checker can verify automatically that these




(a) Behavioral program of class `Snapshot`.




(b) Behavioral program of class `BSnapshot`. It is similar to `Snapshot` with a refined `inactive` state, a local event `success`, and the possibility of launching `regenerate` from the `inactive` state. Restriction `BSnapshot` $\backslash A_{\text{Snapshot}}$ is obtained by removing states and transitions displayed with thick lines.

**Figure 3: Behavioral programs of classes `Snapshot` and `BSnapshot`.**

two formulae are true. Conversely, if a formula is false, the model checker usually gives a counter-example.

## 5. RELATED WORK AND DISCUSSION

Modeling component behavior and protocols and ensuring correct use of component frameworks through a proof system is a recent research line. Most approaches concentrate on the composition problem [14, 1, 8] whereas we are focusing this paper on the substitutability issue.

Most works in the field of Software Architecture for modeling behavior [3] address component compatibility and adaptation in a distributed environment and are often based on process calculi [18, 22, 19]. Some authors put a specific emphasis on the substitutability problem [13]. For instance [4] proposes static subtype checking relying on Nierstrasz's notion of regular types [18]. As another example, in [5], the authors focus on inheritance and extension of behavior, using the $\pi$-calculus as their formal model. These works also consider a distributed environment. The problems of compatibility and substitutability are also significant in fields other than Software Engineering, such as hardware modeling and design. As a matter of example, [6] proposes a "game view" of (hardware) components, relying on deterministic automata.

As far as the objectives (well-formedness, verification, compatibility, and refinement) and models (deterministic automata, non-distributed environment) are concerned, our work is close to the one in [6], although our target applications are similar to the Software Architecture community ones. Another approach introduces behavioral compatibility relying on type-theory [2]. It is more general than ours in its objectives, although quite similar as far as behavioral description is concerned; it is also more general theoretically speaking, while we focus on providing operational tools. In contrast with these works, we restrict to the problem of substitutability in a non-distributed world. Indeed this is what we needed for BLOCKS. Again, this restriction allows us to adopt models more familiar to software developers (UML StateCharts-like), easier to handle (deterministic systems), efficient for formal analysis (model-checking and simulation), and for which there exist effective algorithms and tools. The Synchronous Paradigm [9] offers good properties and tools in such a context. This is why we could use it as the foundation of our model.

As already mentioned our notion of substitutability guarantees the stability of interesting (safety) properties during the derivation process. Hence, at the user level as well as at the framework one, it may be necessary to automatically verify these properties. To this end, we have chosen model checking techniques. Indeed, model checkers rely on verification algorithms based on the exploration of a state space and they can be made automatic since tools are available. They are robust and can be made transparent to framework users. The problem with model checkers is the possible explosion of the state space. Fortunately, this problem has become less limiting over the last decade owing to symbolic algorithms. Furthermore, taking advantage of the structural decomposition of the system allows modular proofs on smaller (sub-)systems. This requires a formal model that exhibits the *compositionality property*, which is the case for our model (theorems 1 and 2).

## 6. CONCLUSION AND PERSPECTIVES

The work described in this paper is derived from our experience and aims at simplifying the correct use of a framework. We have adapted framework technology to the design of knowledge-based system engines and observed a significant gain in development time. For instance, once the analysis completed, the design of a new planning engine based the BLOCKS framework took only two months (instead of about two years for a similar former project started from scratch) and more than 90 % of the code reused existing components [17]. While performing these extensions, we realized the need to formalize and verify component protocols, especially when dealing with subtyping. The corresponding formalism, the topic of this paper, has been developed in parallel with the KBS engines. As a consequence of this initial work, developing formal description of BLOCKS components led us to a better organization of the framework, with an architecture that not only satisfies our design rules but also makes the job easier for the framework user to commit to these rules.

Our behavioral formalism relies on a mathematical model, a specification language, and a semantic mapping from the language to the model. The model supports multiple levels of abstraction, from highly symbolic (just labels) to merely operational (pieces of code).

Moreover, this model is original in the sense that it can cover both static and dynamic behavioral properties of components. To use our formalism, the framework user has only to describe behavioral programs, by drawing simple StateCharts-like graphs with a provided graphic interface. The user may be to a large extend oblivious of the theoretical foundations of the underlying models and their complexity. The model has also a pragmatic outcome: it allows simulation of resulting applications and generation of code, of run-time traces, and of run-time assertions.

Our aim is to accompany frameworks with several kinds of dedicated tools. We are working on tools for manipulating behavioral programs. Currently, we provide a graphic interface to display existing descriptions and modify them. In the future, the interface will watch the user activity and warn about possible violation of the design rules. Since these rules are just sufficient, it is possible for the user not to apply them or to apply them in such a way that they cannot be clearly identified. To cope with this situation, we shall also provide a static substitutability analyzer, based on our model (section 3.1) and a usual partitioning simulation algorithm.

At the present time we have designed a complete interface with *NuSMV* [7], in both directions. First, our description language can be translated into *NuSMV* specifications, and our tool provides also a user friendly way to express the properties the users may want to prove. Second, *NuSMV* diagnosis and return messages are displayed in a readable form: users can browse the hierarchies of behavioral derivations and follow the steps of the proofs. It took us a few weeks to connect our behavioral description language to the *NuSMV* model-checker. The next step is to implement the substitutability analysis tool.

Another interesting feature would be to provide an automatic code generation facility as well as run-time checks. Indeed the behavioral description is rather abstract and may be interpreted in a variety of ways. In particular, automata and associated labels can be given a code interpretation. The generated code would provide skeletal implementations of operations. This code will be correct, by construction–at least with respect to those properties which have been previously checked. Furthermore, the generated code can also be instrumented to provide run-time traces and assertions built in the components.

Developing such tools is a heavy task. Yet, as frameworks are becoming more popular but also more complex, one cannot hope using them without some kind of active assistance, based on formal modeling of component features and automated support.

## 7. REFERENCES


[1] F. Achermann and O. Nierstrasz. Applications = Components + Scripts - A Tour of Piccola. In Mehmet Aksit, editor, *Software Architectures and Component Technology*, pages 261–292. Kluwer, 2001.

[2] S. Alagic and S. Kouznetsova. Behavioral Compatibility of Self-Typed Theory. In B. Magnusson, editor, *ECOOP 2002*, number 2374 in LNCS, pages 585–608, Malaga, Spain, 2002. Springer.

[3] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans, on Software Engineering and Methodology*, 6(3):213–249, 1997.

[4] S. Butkevich, M. Renedo, G. Baumgartner, and M. Young. Compiler and Tool Support for Debugging Object Protocols. In *Proc. 8th ACM SIGSOFT Int. Symp. Foundations of Software Engineering*, pages 50–59, San Diego, CA, USA, 2000. ACM Press.

[5] C. Canal, E. Pimentel, and J. M. Troya. Compatibility and inheritance in software architectures. *Science of Computer Programming*, (41):105–138, 2001.

[6] A. Chakrabarti, L. de Alfaro, T. A. Henzinger, and Freddy Y. C. Mang. Synchronous and Bidirectional Component Interfaces. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proc. CAV*, number 2404 in LNCS, pages 214–227. Springer, 2002.

[7] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: an OpenSource Tool for Symbolic Model Checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proc. CAV*, number 2404 in LNCS, pages 359–364. Springer, 2002.

[8] J. Costa Seco and L. Caires. A Basic Model of Typed Components. In Elisa Bertino, editor, *ECOOP 2000*, number 1850 in LNCS, pages 108–128. Springer, 2000.

[9] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic, 1993.

[10] D. Harel and O. Kupferman. On object systems and behavioral inheritance. *IEEE Trans. Software Engineering*, 28(9):889–903, 2002.

[11] M.R. Henzinger, T.A. Henzinger, and P.W. Kopke. Computing simulation on finite and infinite graphs. *Proc. IEEE Symp. Foundations of Computer Science*, pages 453–462, 1995.

[12] E. M. Clarke Jr., O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[13] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Trans. on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[14] K. Mani Chandy and M. Charpentier. An experiment in program composition and proof. *Formal Methods in System Design*, 20(1):7–21, January 2002.

[15] F. Maraninchi. Operational and Compositional Semantics of Synchronous Automaton Composition. In *Proc. Concur. 1992*, number 630 in LNCS. Springer, 1992.

[16] R. Milner. An algebraic definition of simulation between programs. *Proc. IJCAI*, pages 481–489, 1971.

[17] S. Moisan, A. Ressouche, and J-P. Rigault. BLOCKS, a Component Framework with Checking Facilities for Knowledge-Based Systems. *Informatica*, 25:501–507, 2001.

[18] Nierstrasz O. *Object-Oriented Software Composition*, chapter Regular Types for Active Objects, pages 99–121. Prentice-Hall, 1995.

[19] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Trans. on Software Engineering*, 28(11), Nov 2002.

[20] A. Ressouche and S. Moisan. A Behavior Model of Component Frameworks. Technical report, INRIA, August 2003. to appear.

[21] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison Wesley, 1998.

[22] D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Trans. on Programming Languages and Systems*, 19(2):292–333, March 1997.

# An Assertion Checking Wrapper Design for Java

Roy Patrick Tan
Department of Computer Science
Virginia Tech
660 McBryde Hall, Mail Stop 0106
Blacksburg, VA 24061, USA
rtan@vt.edu

Stephen H. Edwards
Department of Computer Science
Virginia Tech
660 McBryde Hall, Mail Stop 0106
Blacksburg, VA 24061, USA
edwards@cs.vt.edu

## ABSTRACT

The Java Modeling Language allows one to write formal behavioral specifications for Java classes in structured comments within the source code, and then automatically generate run-time assertion checks based on such a specification. Instead of placing the generated assertion checking code directly in the underlying class bytecode file, placing it is a separate wrapper component offers many advantages. Checks can be distributed in binary form alongside the compiled class, and clients can selectively include or exclude checks on a per-class basis without recompilation. In this approach, when checks are excluded the underlying code is just as efficient as if assertions were "compiled out." In addition, a flexible mechanism for enabling or disabling assertion execution on a per-class or per-package basis is also included. This paper presents a design for separating assertion checking code into wrapper classes and discusses the issues arising from this design.

## Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*programming by contract, assertion checkers, class invariants*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*pre- and post-conditions, invariants, assertions*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*object-oriented programming*; D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids*; D.3.2 [**Programming Languages**]: Language Classifications—*JML*

## General Terms

Languages

## Keywords

JML, run-time checking, design by contract, interface violation

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.
*SAVCBS* '03 Helsinki, Finland

## 1. INTRODUCTION

The Java Modeling language (JML) [6] is a behavioral specification language for Java that allows programmers to add model-based specifications to their code. Specifications, including preconditions, postconditions, and invariants, are placed in specially-formatted, structured comments. The JML tool set allows run-time checks to be generated from such specifications and embedded directly in the generated class file, to be checked at run-time [1]. JML's design-by-contract support provides specific syntactic slots that clearly separate the implementation details from the assertion checks. Its support for model-only fields and methods cleanly supports reasoning about a component's abstract state [2].

The benefits of checking design-by-contract assertions are well known [3, 7]. However, due to performance concerns, it is current practice to include run-time assertion checks during testing, but then remove them when distributing production components. This benefits the original implementor, but does little for the clients of that component. As commercial components become more prevalent, and new designs more frequently make use of classes and subsystems purchased from other sources, it is important to consider how such assertion checks can be of use to component clients, as well as how they might add value to a component being offered for sale.

JML-based assertion checks, like those produced by most other techniques, can be left in the compiled, binary version of a class that is distributed to customers. As with other techniques, execution of these checks at run-time can be controlled through a global switch. However, even when checks are not being executed, the resulting code still suffers a performance penalty, both due to the code bloat imposed by the inactive checks and to the cost of constantly looking up whether or not to perform each check.

This paper discusses ongoing work that will address these issues. The goals of this work include:

- Allowing binary distribution of compiled checks alongside the underlying class, so that checks can be included or excluded without source code access or recompilation.
- Imposing no additional overhead when code is run directly, without including the assertion checking wrappers.
- Supporting per-class or per-package run-time enabling or disabling of assertion check execution.

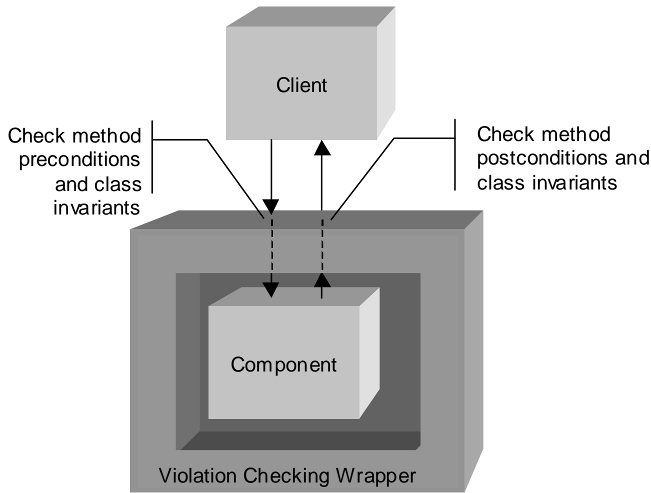



**Figure 1: The wrapper implements assertion checks and delegates other work to the wrapped component.**

- Maintaining transparent compatibility–JML users will not have to change any existing JML source code or the way they compile their code.

## 2. A DESIGN FOR ASSERTION CHECKING WRAPPERS

Our approach adopts a wrapper design [4] that begins with a simple idea: move assertion checking code to a separate class, such that we now have two classes that have the same externally visible features: an unwrapped original class, and the wrapper class that performs the checks but delegates actual computation to the wrapped component. Figure 1 illustrates this approach.

In principle, the concept of using a wrapper to separate assertion checks from the underlying component is simple. Following the decorator pattern [5], the wrapper provides the same external interface as the underlying component, and just adds extra features transparently. One can then extract a common interface capturing the publicly visible features of the underlying component, and set up the wrapper and the original component as two alternative implementations of this interface. By using a factory method [5], the decision of which implementation object to instantiate at any given point can be separated from the object requesting the instance. Using a factory shifts control over which instance to create into another component. With the right support, this also allows users to enable or disable assertion checks on a per class basis, at run-time.

```
public class List
{
    private /*@ spec_public @*/ int elementCount;

    //@ requires elementCount > 0;
    public Object removeFirst() {
        // implementation here ...
    }

    // ...
}
```

**Figure 2: A `List` class, abbreviated for simplicity.**

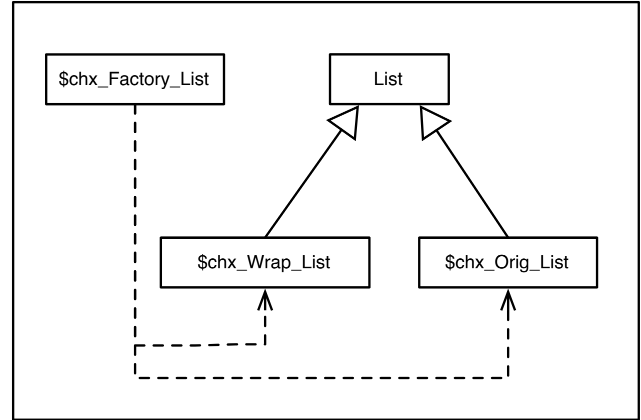



**Figure 3: The original list class will be transformed into four components.**

The implementation details are probably best illustrated with an example. Figure 2, shows a snippet of code for a simple `List` class. For this paper, we are not interested in how the methods (such as `removeFirst`) are implemented, so we do not show implementation code. What we are interested in is how to separate the assertions (as illustrated by the `requires` clause) from the implementation code.

As shown in Figure 3, the wrapper-based design involves automatically generating four different class files from the source shown in Figure 2:

- The original class that contains the actual implementation.
- The wrapper class that contains the assertion checks.
- An interface that both the original and wrapper classes implement.
- A factory class that is called to create an instance of `List`.

```
public interface List
{
    // ...
    public int $chx_get_elementCount();

    public void $chx_set_elementCount(int count);

    public Object removeFirst();
}
```

**Figure 4: Instead of a class, there is now a `List` interface that both the wrapper and nonwrapper classes implement.**

```
public class Wrapper
{
    public Wrappable wrappedObject = null;
    public static CheckingPrefs isEnabled = null;
}

public class $chx_Wrap_List extends    Wrapper
                            implements List
{
   // ...
   public int $chx_get_elementCount() {
      return wrappedObject.elementCount();
   }

   public Object removeFirst() {

      // ...

      if ( isEnabled.precondition() ) {
         // checkPre performs the actual
         //precondition check.
         checkPre$RemoveFirst$List();
      }

      return (($chx_Orig_List)wrappedObject)
                  .removeFirst();
   }
}
```

**Figure 5: The `Wrapper` class, from which all wrappers inherit, and the generated wrapper class for `List`.**

Both the wrapper and the original class perform the same essential operations—both export the same (behavioral) interface. In our design, we make this explicit by making `List` an interface and having both the wrapper class and the original class implement it. Figure 4 shows a snippet of this automatically generated `List` interface. The `List` interface redeclares the public methods of the original class. Also, accessor method declarations for public fields are added so that the fields accessible in the original class are also accessible through the interface.

The wrapper class "wraps" or decorates an instance of the original component, but adds checking code before and after every method. To do this, every wrapper component has a `wrappedObject` field to hold a reference to the wrapped instance of the original component. This is achieved by having every wrapper component be a subclass of `Wrapper`. Then, each method defined in the original component is also defined in the wrapper, where it is implemented by performing any pre-state checks, delegating to the wrapped object for the core behavior, and then performing any post-state checks.

Figure 5 shows the `Wrapper` class from which all wrappers inherit. It contains just two fields, `wrappedObject` and `isEnabled`. The `isEnabled` member can be queried to determine whether or not particular checks should be performed at run-time. Both members are initialized by the factory method that creates instances of the corresponding wrapper class.

Figure 5 also illustrates the basic structure of the wrapper

```
public class $chx_Statics_List
{
   public static CheckingPrefs isEnabled = null;
   public static List newObject() {
      List      result    = new $chx_Orig_List();
      Wrappable wrappable = (Wrappable)result;
      if ( isEnabled != null && isEnabled.wrap() )
      {
         result = new
             $chx_Wrap_List( result, isEnabled );
      }
      wrappable.$chx_this = result;
      return result;
  }
}
```

**Figure 6: A factory is used to create `List` instances.**

class for `List`; it shows the output of our wrapper generator tool for JML, simplified for brevity. Before every assertion check is performed, the `isEnabled` field is queried by calling the appropriate method. In Figure 5 for example, `isEnabled.precondition()` is tested and the precondition is checked only if the method returns `true`. An exception is thrown if an executed check fails.

The factory method that is invoked to create new `List`s is shown in Figure 6. For every constructor in the original class, there is a corresponding factory method in the factory class. If every `List` object is instantiated using the factory instead of a call to `new`, we can transfer the decision of whether to create a wrapped or unwrapped version of the object to a separate component. In this case, the factory method queries the static member `isEnabled` to decide whether to return a wrapped or unwrapped object.

So far, our discussion has dealt with transforming only one class. So how is inheritance addressed in this technique? For example, if `List` inherits from `Bag`, what relationship(s) exist between the generated wrapper classes, generated interfaces, and original classes? The solution is straightforward: let each of `List`'s interface, wrapper, and original class extend the corresponding interface or class from `Bag`. That is, the `List` wrapper extends the `Bag` wrapper, the `List` interface extends the `Bag` interface, and the original `List` class extends the original `Bag` class. Figure 7 illustrates this idea.

At the highest level of the inheritance hierarchy, instead of the wrapper class inheriting directly from `Object`, the wrapper class inherits from `Wrapper`. Similarly, the non-wrapper class inherits from `Wrappable`. A practical implication of this design is that if a class has JML specifications, all of its superclasses must be transformed by our tool regardless of whether they have specs or not. This process can be handled automatically by extending the JML tool set. Even if source files for the superclass(es) without specifications are unavailable, we can obtain the needed signature of the class through reflection or inspection of the bytecode.

# 3. ELIMINATING THE CHANGES NEEDED IN CLIENT CODE

Placing assertion checks in wrappers provides several advantages: assertion checks can be selectively included or excluded without requiring recompilation, and when they are

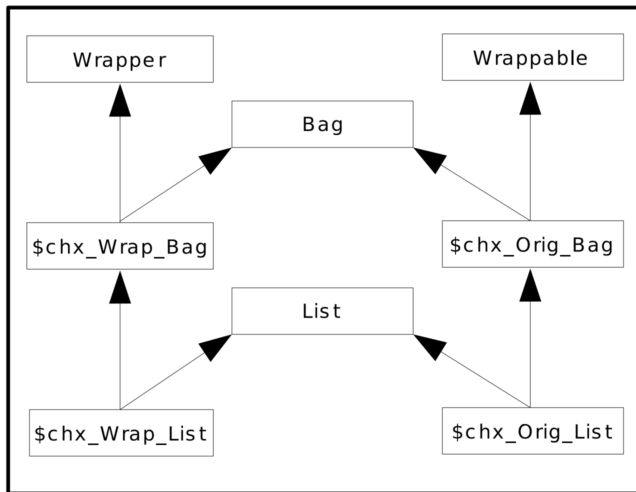



**Figure 7: Dealing with inheritance: `List` inherits from `Bag`.**

```
public class $chx_Orig_List extends Wrappable
{
    private int elementCount;

    public int $chx_get_elementCount() {
        return elementCount;
    }

    // ...

    public Object removeFirst() {
        // implementation here ...
    }

    // ...
}
```

**Figure 8: The original class is modified.**

excluded there is no additional overhead imposed. On the other hand, the design presented in Section 2 requires some changes to basic coding practices in both the client code and in component being checked.

## 3.1 Changes to Client Code

The changes needed in client code fall into three areas: object creation, data member access, and static method invocation. Rather than calling `new` to create a new object, client code must now call the corresponding factory method. For example:

```
List l = new List();
```

now should be phrased this way

```
List l = $chx_Statics_List.newObject();
```

In addition, the way public fields are accessed changes. Current Java design practices discourage the use of publicly accessible data members. However, for code that violates such practices, within this framework there can never be direct access to such fields. Instead, automatically generated accessor methods must be used:

```
a = l.length;
```

now should be phrased this way

```
a = l.$chx_get_length();
```

Finally, static method calls to the class being checked must also be transformed. The `$chx_Statics_List` class generated by the tool set will also contain a dispatch stub for each static method in the class being wrapped. The stub will, depending on whether or not wrapper usage is enabled, forward the call to a corresponding static method in either the wrapper or in the underlying class.

## 3.2 Changes to the Original Class

The original class also needs modification to work within this framework. First, since we've appropriated the name `List` for the interface, we rename the original `List` class to `$chx_Orig_List`. Second, private methods within the class are promoted to package level access so that the wrapper can have access to these methods (to adding checks to them). Third, accessor methods for all data members must be generated. Figure 8 illustrates these modifications on the original `List` class.

One problem with this approach is that when the unwrapped object calls another method of its own, that call will not go through the wrapper so its behavior will not be checked. That is, if `removeFirst` calls a method belonging to `this`, the method must be checked for contract violations as well. The solution is for the original class to inherit from an object called `Wrappable`, which contains one field: `$chx_this`. The `$chx_this` field is a reference to either the associated wrapper if the non-wrapper object is contained inside one, or to `this` if it is not wrapped. Each time the original class invokes one of its own methods, instead of using `this` (e.g. `this.m()`), the modified version of the original class uses `$chx_this` (e.g. `$chx_this.m()`).

A related problem is calling non-public methods. If a method has a call to some private method `this.p()`, the call would be translated to `$chx_this.p()` where `$chx_this` might be a wrapper object. However, to perform the actual computation, the wrapper object must also be able to access the wrapped object's original method. Thus, private methods must be elevated to at least package level access. A similar problem exists for protected methods, but they must be promoted to public access, since superclasses may not be in the same package as the subclass. This means that certain access control violations may not be caught at runtime. Violations should still be detectable at the compile phase, however.

## 3.3 Removing the Need for Source-Level Modifications

All of these modifications, both to the client code and to the component being checked, add clutter and complexity. Further, if we wish for wrapper-based objects to be used by

existing code, perhaps code distributed in binary-only form, then how can we impose stylistic modification requirements on that code? Finally, changes to both the client code and the component code to adopt this framework will necessarily impose additional overhead, even when check execution is disabled.

To address these concerns, we are designing a custom class loader for the JML tool set. Rather than requiring the client code and the underlying component to have modifications transformed into them at compile time, instead the class loader can dynamically transform the original bytecode sequences at load time if the wrapper framework is being used.

In essence, the JML compiler generates bytcode for the original `List` class in the file `List.class`, just as if no assertions were being used. The bytecode for the three other wrapper support classes are generated in `*.chx` files. One can run the resulting Java program normally using the `java` command, which has the effect of completely ignoring all of the wrapper-related files and running the original unmodified bytecode. Alternatively, by adding the assertion checking class loader at the start of the command line, the necessary modifications to both client and component code are made on-the-fly at load time. This class loader knows about the special file name extension used by the wrapper support classes so that it can detect and load wrapper-enabled classes differently than code without JML assertions. This approach allows the wrappers to be distributed in binary form along with the original component, but still maintain zero additional overhead when wrappers are not being used.

We have designed such a class loader and are in the process of implementing it. After exploring the critical security issues, it also appears possible to use this technique to retroactively add wrapper-based assertion checking features even within protected packages, such as `java.util`.

# 4. RUN-TIME CONTROL OF ASSERTION BEHAVIOR

As with most competing techniques, JML allows embedded assertion checks to be enabled or disabled when the code is run. Effectively, JML uses a single global switch for each family of assertions—so postconditions can be disabled independently of preconditions, for example. Some tools provide more fine-grained control. iContract provides a graphical interface for selectively enabling or disabling the generation and inclusion of assertion checks on a per-class basis at build time.

With the wrapper approach, inclusion or exclusion of assertion checks is deferred until load time. As a result, it would be more preferable to allow fine-grained control over which wrapper-enabled classes should use wrappers, as well as which families of assertion checks should actually be executed at program startup or even during run-time. We have devised an approach that allows this control at the individual class level, as well as at the Java package level.

As discussed in Section 2, every wrapper is given a reference to a `CheckingPrefs` object called `isEnabled`. Rather than using a single global object for this purpose, there is one such `CheckingPrefs` object for every wrappable class. Further, there is a similar object for each Java package, and the package-level and class-level preferences objects are linked together into a tree structure that mirrors the Java package nesting structure. In this tree, the preferences objects for individual classes are the leaves. The custom class loader takes care of incrementally constructing this tree as classes are loaded. Note that when an individual wrapper checks a preference setting via `isEnabled`, it retrieves the setting directly from that object and no tree traversal is needed.

Using a graphic control panel at program startup, one can use a collapsible/expandable tree view to set preferences about what whether or not wrappers should be used on wrappable classes, and if wrappers are used, which assertions should be executed and which should be skipped. Tree nodes in this graphical view map directly to the tree-structured network of preferences objects. If one changes an option, that change is stored in the corresponding preferences object and also propagated down through its children all the way to the leaves. Thus, tree traversal happens when the user makes an option setting, rather than when settings are looked up inside each assertion test.

In addition to making such changes at program startup, the same control panel could also be used to modify preferences during run-time by executing it in a separate thread. Further, preference settings could also be saved to or read from properties files. The result is a flexible approach to fine-grained control of assertion checking options that scales well.

# 5. CONCLUSIONS AND FUTURE WORK

In this paper, we have outlined a strategy for extending JML assertion checking using wrappers. Using wrappers allows checking code to be distributed in compiled form alongside the original code, eliminates the associated overhead of checking code when it is unused, and allows run-time control over contract checking on a per-class basis.

This approach is not without challenges, however. Two of the more troublesome are dealing with inline assertion checks and dealing with `super` calls. The wrapper approach deals only with assertions that can be checked before and after a method is called. Assertions within methods need to be handled in a different manner. One possible solution is to place additional functionality into the custom classloader such that it can inject the appropriate assertion checks into the bytecode when the class is being loaded.

The problem with `super` calls is that when within a component it calls a method of its superclass, there is not an easy way to call the superclass of the wrapper component instead, and so in our current design these calls go unchecked. JML currently solves a similar problem by renaming methods and using reflection to call the superclass methods. The disadvantage of this approach is the high performance overhead of reflection in Java.

We are in the process of completing the three parts needed to make this framework viable. First, the JML compiler has been extended to take in JML-annotated Java source code and produce the four corresponding files described in Section 2. The original class's bytecode should be the same as if it were compiled with `javac`. Second, the custom classloader for incorporating load-time modifications to client and component code has been designed and must be completed. Third, a controller that allows users to manage assertion execution preferences in a convenient way has been prototyped and must be completed.

## Acknowlegements

We gratefully acknowledge the financial support from the National Science Foundation under the grant CCR-0113181. Any opinions, conclustions or recommendations expressed in this paper do not necessarily reflect the views of the NSF.


## 6. REFERENCES


[1] Y. Cheon. A runtime assertion checker for the java modeling language. Technical Report 03–09, Department of Computer Science, Iowa State University, April 2003.

[2] Y. Cheon, G. T. Leavens, M. Sitaraman, and S. H. Edwards. Model variables: Cleanly supporting abstraction in design by contract. Technical Report 03–10, Department of Computer Science, Iowa State University, March 2003.

[3] S. H. Edwards. Black-box testing using flowgraphs: an experimental assessment of effectiveness and automation potential. *Software Testing, Verification & Reliability*, 10(4):249–262, 2000.

[4] S. H. Edwards. Making the case for assertion checking wrappers. In *Proceedings of the RESOLVE Workshop 2002*, pages 95–104. Dept. of Computer Science, Virginia Tech, 2002.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[6] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[7] J. M. Voas. Quality time: How assertions can increase test effectiveness. *IEEE Software*, 14(2):118–119, 1997.

# An Approach to Model and Validate Publish/Subscribe Architectures

Luca Zanolin, Carlo Ghezzi, and Luciano Baresi
Politecnico di Milano
Dipartimento di Elettronica ed Informazione
P.za L. da Vinci 32, 20133 Milano (Italy)
{zanolin|ghezzi|baresi}@elet.polimi.it

## ABSTRACT

Distributed applications are increasingly built as federations of components that join and leave the cooperation dynamically. Publish/subscribe middleware is a promising infrastructure to support these applications, but unfortunately complicates the understanding and validation of these systems. It is easy to understand what each component does, but it is hard to understand what the global federation achieves.

In this paper, we describe an approach to support the modeling and validation of publish/subscribe architectures. Since the complexity is mainly constrained in the middleware, we supply it as a predefined parametric component. Besides setting these parameters, the designer must provide the other components as UML *statechart diagrams*. The required global properties of the federation are then given in terms of *live sequence charts* (LSCs) and the validation of the whole system is achieved through model checking using SPIN. Instead of using the property language of SPIN (*linear temporal logic*), we render properties as automata; this allows us to represent more complex properties and conduct more thorough validation of our systems.

## 1. INTRODUCTION

The *publish/subscribe paradigm* has been proposed as a basis for middleware platforms that support software applications composed of highly evolvable and dynamic federations of components. According to this paradigm, components do not interact directly, but their communications are mediated by the middleware. Components declare the events they are interested in and when a component publishes an event, the middleware notifies it to all components which subscribed to it.

Publish/subscribe middleware decouples the communication among components. The sender does not know the receivers of its messages, but it is the middleware that identifies them dynamically. As a consequence, new components can dynamically join the federation, become immediately active, and cooperate with the other components without any reconfiguration of the architecture.

The gain in flexibility is counterbalanced by the difficulty for the designer to understand the overall behavior of the system. It is hard to get a picture of how components cooperate and understand the global data and control flows. Although components might be working correctly when they are examined in isolation, they could provide erroneous services in a cooperative setting.

These problems motivate our approach to model and validate publish/subscribe architectures. Modeling the middleware is the most complex task, since we must consider how components communicate in a distributed environment. The complexity of such a model is partially counterbalanced by the fact that the middleware is not application-specific. We can use the same (model of a) middleware for several different architectures. This is why we provide a ready-to-use parametric model of the middleware. The designer has to configure it and provide the other components as *UML statechart diagrams* [17]. Following this approach, designers trust the middleware and validate the cooperation of components.

The designer describes the global properties of the federation in terms of *live sequence charts* (LSCs) [6] and the validation of the whole system is achieved through model checking using SPIN [10]. LTL (linear temporal logic), the property language of SPIN, is not enough to render what we want to prove. We by-pass this limitation by transforming LSCs in automata. Both the model and properties are then translated into Promela [9] and passed to SPIN.

The paper is organized as follows. Section 2 presents our approach to model publish/subscribe architectures and exemplifies it through a simple example of a hypothetical *eHouse*. Section 3 discusses validation and shows how to model the properties for our case study and how to transform them into automata. Section 4 surveys the related work, and Section 5 concludes the paper.

## 2. MODELING

In publish/subscribe architectures, components *exchange* events through the middleware. Thus, any model of these architectures must explicitly consider the three main actors: events, middleware, and components. The next three sections describe how to model them using our approach and exemplify all concepts on a fragment of a hypothetical *eHouse*. We focus on a simple service that allows users to

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.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

take baths. When the user requires a bath, the service reacts by warming the bathroom and starting to fill the bath tub. When everything is ready, the user can take the bath. Lack of space forbids us to present all models, but interested readers can refer to [13] for the complete set.

## 2.1 Events

Publish/subscribe architectures do not allow components to exchange events directly. The communication is always mediated by the middleware through the following operations: *subscribe*, *unsubscribe*, *publish*, and *notify*. Components *subscribe* to declare the interest for some specific events, and, similarly, they *unsubscribe* to undo their subscriptions. Components *publish* events to the middleware that *notifies* them to the other components.

Events have a name and a (possibly empty) set of parameters. When components subscribe/unsubscribe to/from events, they can specify them fully or partially. For instance:

*subscribe("bath", "ready")*

means that the component wants to know only when the bath is ready, but:

*subscribe("bath", $)*

means that it wants to know all *bath* events.

## 2.2 Middleware

As we have already said, the developer does not model the middleware explicitly, but has to configure the *middleware component* that we supply with our approach. Unfortunately, the term middleware is not enough to fully characterize its behavior. The market offers different alternatives: standards (e.g., Java Message Service (JMS) [19]) and implementations from both university (i.e., Siena [1], Jedi [4]) and industry (i.e., TIBCO [20]). All these middleware platforms support the publish/subscribe paradigm, but with different qualities of service.

Given these alternatives, the definition of the parameters that the middleware component should offer to developers is a key issue. On one hand, many parameters would allow us to specify the very details of the behavior, but, on the other hand, they would complicate the model unnecessarily. The identification of the minimal set of parameters is essential to verify the application: the simpler the model is, the faster verification would be. Since our goal is the verification of how components cooperate, we can assume that the model must describe what is seen by components and we can get rid of many internal details. For instance, the middleware can be distributed on several hosts, but this is completely transparent to components and provided services are the same. Important differences are on the quality of service, thus we model how services are provided, instead of modeling how the middleware works.

After several attempts, we have selected the following three characteristics as the key elements that concur in the definition of the behavior of a generic middleware:

**Delivery** Some middleware platforms are designed to cope with mobile environments or with strong performance on the delivery of events. Due to these requirements, the designer can choose to lower the warranties on delivery of events and not to guarantee that all published events are notified to all components. On one hand, this reduces the reliability of the system, but, on the other hand, it increases its performance. Thus, event delivery can be characterized by two alternatives: (a) all events are always delivered, or (b) some events can be lost.

**Notification** If a middleware behaves *correctly*, it notifies events by keeping the same order in which they were published. Nevertheless, this behavior is not always respected due to the environment in which the middleware executes. If we want to relate the order of publication to the order of notification, we can identify three alternatives: (a) they are the same, (b) the order of publication and notification are the same only when we refer to the events published by the same component, or (c) there is no relationship and events may be notified randomly.

For instance, if component $A$ publishes events $x_1$ and $x_2$, and component $B$ publishes $y_1$ and $y_2$, the middleware could notify these events to component $C$ as follows:

- case (a), $x_1 < x_2 < y_1 < y_2$
- case (b), $x_1 < x_2$ , $y_1 < y_2$
- case (c), any permutation.

where $x < y$ means that $x$ is notified before $y$.

Note that the first hypothesis only works in general in a centralized setting. It can be used as an idealized approximation in other practical cases.

**Subscription** When a component declares the events in which it is interested, that is, it subscribes to these events, it starts receiving them. However, the distributed environment can make the middleware not to react immediately to new subscriptions. Once more, our characterization identifies two alternatives: (a) the middleware immediately reacts to (un)subscriptions, or (b) these operations do not have immediate effects and are delayed.

The actual middleware comes from choosing one option for these three main characteristics. These characterizations cover most of the warranties that a middleware should satisfy. If this is not the case, developers can always get rid of parametric middleware, elaborate their particular model of the middleware, as a UML statechart diagram, integrate it in the architecture, and validate the whole system. They lose the advantages as to modeling, but keep the validation approach.

To increase the confidence in the parametric model of the middleware, we validated it by applying the same approach that we are proposing in this paper. We built a federation of simple – dummy – components to stress all the possible configurations of our middleware. The ease of components allowed us to state that any misbehavior was due to the middleware itself.

Referring to the eHouse example, we assume that the architecture is built on a middleware that delivers all events, keeps the same order of publication and notification, and reacts immediately to all (un)subscriptions.

## 2.3 Components

The designer provides a UML statechart diagram for each component, where transitions describe how the component reacts upon receipt of an event. Transition labels comprise two parts separated by `/`. The first part (i.e., the precondition) describes when the transition can fire, while the second part defines the actions associated with the firing of the transition. For instance, the label:

*consume("bath", "full") / publish("bath", "ready")*

states that the transition can fire when the component is notified that the bath tub is full of water and publishes an event to say that the bath is ready.

Notified events are stored in a *notification queue*. The component retrieves the first event and if it does not trigger any transition exiting the current state, the component discards it and processes the following one. This mechanism allows components to evolve even if they receive events in which they are not interested in their current state.

Moving to our running example, besides the middleware described in the previous section, we have five components that cooperate to provide the service: *User*, *Bathroom*, *Heating*, *Bath*, and *PowerManager*.

*User* publishes events to notify that she/he wants to take a bath. *Bathroom* is in charge of setting the bath and increasing the temperature in the bathroom. The bath and heating system are described by *Bath* and *Heating*, respectively. When *Bath* receives the event from *Bathroom*, it starts operating and publishes another event when the bath tub is full. At the same time, *Heating* turns on the electric heating and increases the temperature of the bathroom.

Finally, *PowerManager* manages the provision of electricity. If there is a blackout, this component notifies this failure and switches from primary to secondary power supplier. The secondary power supplier is less powerful than the primary one and some electric devices must be turned off. For instance, the electric heating turns itself off as soon as there is a blackout. Thus, the user cannot take a bath: The temperature in the bathroom cannot be increased since the electric heating does not work.

Figure 1 shows the statechart diagram of *Bathroom*. It starts in state *Idle* waiting for some events. At this stage, it is only subscribed to events that ask for a bath. When the user notifies that she/he needs a bath, *Bathroom* evolves and notifies *Heating* that the temperature should be increased and *Bath* should start to run. At the same time, the component updates its subscriptions by adding those about temperature, heating, and bath.

This component exploits two variables that are used to store the status of the bath, *bathStatus*, and of the temperature, *temperatureStatus*. For instance, the variable *bathStatus* set to *true* means that the bath tub is full of water.

When the bath tub is full of water and temperature is hot, the bath is *Ready*. In this state, the component is not interested in the events about bath and heating anymore, thus it unsubscribes from them. Finally, after the user takes the bath, *Bathroom* restores temperature to *cold*[1].

Figure 2 shows the statechart diagram of *Heating*. For simplicity, we suppose that, when this component starts, the power supplier is working correctly and temperature is *cold*. When *Heating* receives an event that asks for increasing the

[1]Temperature can only assume two values: cold and hot.

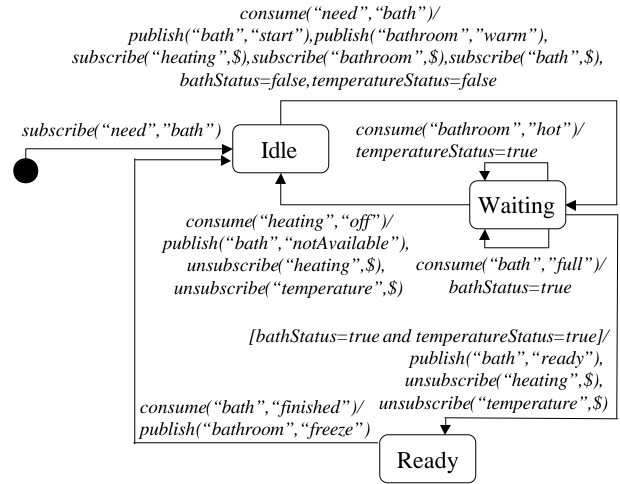



**Figure 1: Bathroom**

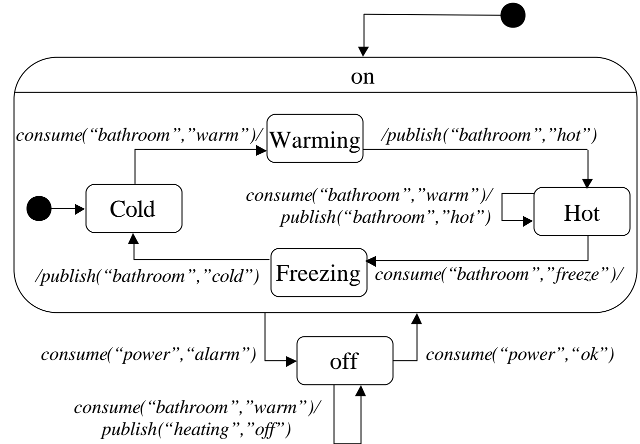



**Figure 2: Heating**

temperature, it moves to an intermediate state to say that the bathroom is warming. When the temperature in the bathroom becomes *hot*, *Heating* moves to the next state, i.e., *Hot*.

# 3. VALIDATION

Validation comprises two main aspects: the definition of the properties to validate the cooperation of components and the transformation of both the model and properties into automata (i.e., Promela).

## 3.1 Properties

Our goal was to provide an easy-to-use graphical language to specify properties, which would allow designers to work at the same level of abstraction as statechart diagrams. For these reasons, we did not use any temporal logic formalisms, like *linear temporal logic* [16] (LTL), since they work at a different level of abstraction and, thus, developers can find it difficult to use. We chose *live sequence charts* (LSCs) [6] since they are a graphical formalism powerful enough to describe how entities exchange messages, that is, the properties that we want to analyze.

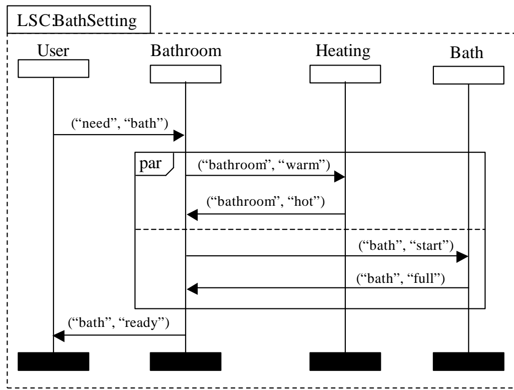



**Figure 3: A basic LSC: Bath Setting**

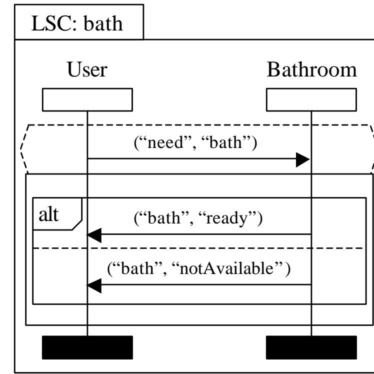



**Figure 4: LSC: Bath Ready**

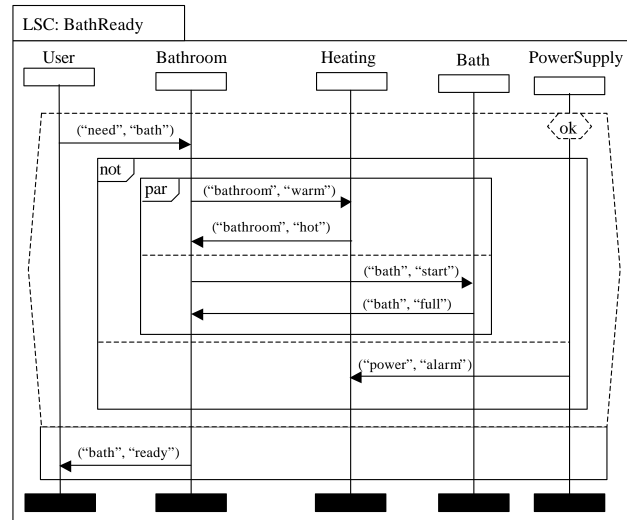



**Figure 5: LSC: Bath Ready (revised)**

Briefly, a basic LSC diagram describes a scenario on how the architecture behaves. LSCs allow us to render both existential and universal properties, that is, scenarios that must be verified in at least one or all the evolutions of the architecture.

Entities are drawn as white rectangles with names above. The life-cycle of an entity is rendered as a vertical line and a black rectangle: The entity is created when we draw the white rectangle and dies when we draw the black one. Messages exchanged between entities are drawn as arrows and are asynchronous by default. Each message is decorated with a label that describes the message itself.

In our approach, we assume publish/subscribe as the underlying communication policy and we omit the middleware in the charts. When we draw an arrow between two entities, we implicitly assume that the message is first sent to the middleware and then routed to the other entity. The arrow means that the target entity receives the notification of the message and that the message triggers a transition inside the entity (i.e., inside its statechart).

After introducing LSCs, let us define some properties on how our bath service should behave. For instance, we want to state that, when the user requires the bath, the temperature in the bathroom increases and the bath tub starts to fill. This two tasks are done in parallel and after the termination of both, the user is notified that the bath is ready. This property is described in Figure 3, which shows a basic LSC scenario. *User* requires the bath and *Bathroom* reacts notifying that *Heating* must start to warm the bathroom and *Bath* to fill the bath tub. The two tasks are performed in parallel without any constraint on their order. This parallelism is described through the *par* operator that states that its two scenarios (i.e., warming the bathroom and filling the bath tub) evolve in parallel without any particular order among the events they contain. When the bath tub is full and the bathroom is warm, *User* is notified that the bath is ready.

This chart describes only a possible evolution since we cannot be sure that *Bathroom* always notifies that the bath is ready. In fact, if we had a blackout, *User* would receive an event to inform her/him that the bath cannot be set. We do not require that the application always complies with this scenario, but, that there are some possible evolutions that are compliant with it. In LSCs, these scenarios are called *provisional* or *cold* scenarios and are depicted in dashed rectangles, as shown in Figure 3.

To fully specify the bath service, the designer also wants to describe the possible evolutions of the service, that is, when the user requires a bath, she/he always receives a positive or negative answer. This property is shown in Figure 4. LSCs allow us to define such a property through a *mandatory* or *hot* scenario. In general, it is difficult to identify global properties that must be satisfied in all evolutions. For this reason, LSCs support the definition of pre-conditions, that is, the property must hold in all the evolutions for which the precondition holds. Preconditions are drawn in dashed polygons, while the hot part of the scenario is depicted in a solid rectangle. For clarification, we can say that the precondition implies the hot scenario. The chart in Figure 4 states that, for all the evolutions in which *User* requires a bath, *Bathroom* notifies two possible events, that is, the bath is ready or the bath is not available. In this chart, we exploit *alt* (alternate), which is another operator supported by LSCs. This operator says that one of its two scenarios must hold. Thus, Figure 4 describes that, when we require a bath, we must receive an event to know if the bath is ready or not.

Finally, we can redefine the previous property (Figure 3) to define when the bath must be available: The bath must always become ready if in the meanwhile we do not have

blackouts. This property is described in Figure 5. If we have no blackouts while *Heating* warms the bathroom, the bath must always become available. In this chart, we introduce the *not* operator that is not part of standard LSCs. This operator has two scenarios as parameters and states that while the first evolves, the second cannot happen simultaneously: If we have no blackouts while the bath sets itself, *User* always receives a positive answer, i.e., the bath is ready.

## 3.2 Transformation

So far, we have shown how to model the architecture and define the properties. After these steps, we can start the analysis. We decided to use the SPIN model checker [10] as verifier, but as we have already explained we do not use LTL to describe the properties that we want to prove. Everything is transformed into automata and then translated into Promela [9].

We customize the middleware according to the parameters set by the developer. Each alternative corresponds to a Promela package and the tool selects the right packages and assembles the model of the middleware directly.

Translating of statechart diagrams in Promela is straightforward. We do not need to describe this translation since it has been done by others before (e.g., vUML [14] and veriUML [2]) and to implement our translation we have borrowed from these approaches.

Properties are translated in two ways: They are described through automata and, if necessary, through auxiliary LTL formulae. This translation is rather complex since SPIN does not support the verification of existential properties natively. It can verify LTL formulae, which define universal properties, but not existential ones.

To state an existential property through LTL, we could negate the LTL formula and verify that the system violates it: This means that there is at least one evolution in which the LTL formula is satisfied (i.e., its negation is violated). However, this approach would require that SPIN be run several times for each property that we want to verify. Instead of using LTL formulae and their translation into Büchi automata, we investigate a different solution based on simple automata rendered as Promela processes.

We verify if LSC hold by reasoning on the state reachability feature provided by SPIN. However, automata are not enough to describe all the LSC features, and, when required, we introduce LTL formulae to overcome this problem. The transformation of an LSC into an automaton (and LTL) goes through these four steps:

1. We simplify the property by downgrading all the hot scenarios to cold ones;

2. We translate the simplified property into an automaton that recognizes the sequence of events described by the LSC. This task is quite easy since the structure of the automaton replicates the one of the LSC;

3. We reintroduce the fact that the scenario is *hot* by identifying the states in the automaton in which the hot scenario starts and ends;

4. We describe the *hot* scenario through constraints expressed as LTL formulae. These constraints state that if an automaton has reached the state that corresponds to the first message of the hot scenario, it must always reach the state that corresponds to the last message of the hot scenario. In other words, if the automaton recognizes the first message of the hot scenario, it must always recognize all the messages that belong to the same hot scenario.

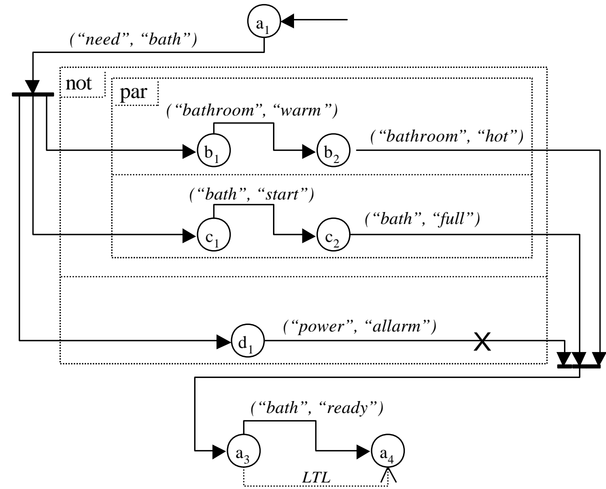



**Figure 6: The automaton that corresponds to the LSC of Figure 5**

The combination of the automaton and LTL formulae allows us to translate any LSC into Promela and verify it through SPIN.

For space reasons, we omit the details of this algorithm and we illustrate the translation informally through an example. Let us consider the LSC of Figure 5 and the corresponding automaton of Figure 6. This automaton has three types of arrows: solid, solid with a cross, and dashed. Solid arrows describe *standard* transitions and are decorated with labels that describe recognized events. For instance, if the automaton evolves from state $b_1$ to state $b_2$, this means that the event *("bathroom", "warm")* has been published and consumed by the component[2]. This type of arrow can end in a state or in a fork/join bar. Besides recognizing events as the previous kind, solid arrows with a cross disable the join bar in which they end. For instance, when the transition leaving $d_1$ fires, the join bar in the right-hand side of the figure is disabled. Finally, dashed arrows do not define transitions between states, but describe constraints on the evolution of the automaton. The constraint – described by an LTL formula – is always the same: If the automaton reaches a state (i.e., the source state of the arrow), it must always reach the other state (i.e., the target state of the arrow).

The automaton of Figure 6 has the same structure as the LSC of Figure 5. This means that when the middleware notifies the first event (i.e.,*("need", "bath")*), the fork bar is enabled and the automaton splits its evolution in three different threads. Moving top-down, the first thread describes the warming of the bathroom, the second thread the filling of the bath tub, and the last thread corresponds to the blackout.

[2]For the sake of clarity, in Figure 6 we do not describe who publishes or consumes events.

If the first two threads evolve completely while the third thread does not, the join bar is enabled and the automaton evolves to state $a_3$. This means that we do not have a blackout while the bathroom is warming and the bath tub is filling. Then, if the middleware notifies the last event, the automaton reaches state $a_4$.

Reasoning on state reachability, we can argue that, if state $a_4$ is reachable, then there is at least one evolution that complies with the simplified property (i.e., the cold scenario). The property described by this automaton – with no LTL formula – states that there exists an evolution in which we have no blackout, after setting, the bath becomes available to the user.

Nevertheless, the property of Figure 5 states that, if we have no blackout, the bath must be always available. This is why we must refine the automaton and add the last arrow. In the example, the hot scenario only concerns the last event, that is the transition between states $a_3$ and $a_4$. The hot constraint means that, if the precondition holds (i.e., all the previous events have already happened), then this event must always occur. This constraint is described by an LTL formula:

$$\Box(In(a_3) \Rightarrow \Diamond In(a_4))$$

where we require that when the automaton is in state $a_3$ (i.e., $In(a_3)$ holds), it must always reach state $a_4$.

We can verify this property by reasoning on the reachability of states. In particular, we require that the final state $a_4$ be reachable, thus there is at least an evolution that is compliant with this property. Finally, if the model checker does not highlight any evolution in which the LTL formula is violated, we can say that when the precondition is verified, it is always the case that the post-condition is verified in the same evolution.

The whole translation – middleware, components, and properties – creates a specification that can be analyzed by SPIN. If it finds misbehaviors in the way components cooperate, they are rendered back to the designer as an evolution of the trace that highlights the error. Unfortunately, this is not always the case: Some misbehaviors cannot be simply described through execution traces. For instance, if a *cold* scenario is not respected, that is, no evolution is compliant with the scenario, it is meaningless to show an execution of the system.

The validation of the bath service has been performed by assuming a middleware that delivers all events, keeps the same order of publication and notification, and reacts immediately to all (un)subscriptions. The validation process shows that the service is incorrectly designed since it violates the property shown in Figure 4. For example, if we consider the following scenario: *User* asks for a bath and *Bath* starts to fill the bath tub. At the same time *Heating* increases the temperature, but before becoming hot, we have a blackout that turns the heating off. At this point, *Bathroom* and *User* wait forever since *Heating* does notify neither that it is switched off nor that temperature is hot.

This misbehavior can be avoided by modifying the arrows between states *on* and *off* in *Heating* (Figure 2) to introduce the publication of an event to notify the failure of *Heating*.

# 4. RELATED WORK

Software model checking is an active research area. A lot of effort has been devoted to applying this technique to the validation of application models, often designed as UML statechart diagrams, and the coordination and validation of the components of distributed applications based on well-defined communication paradigms.

vUML [14], veriUML [2], JACK [7], and HUGO [18] provide a generic framework for model checking statecharts. All of these works support the validation of distributed systems, where each statechart describes a component, but do not support any complex communication paradigm. JACK and HUGO only support broadcast communication, that is, the events produced by a component are notified to all the others. vUML and veriUML support the concept of channel, that is, each component writes and reads messages on/from a channel. These proposals aim at general-purpose applications and can cover different domains. However, they are not always suitable when we need a specific communication paradigm. In fact, if we want to use them with the publish/subscribe paradigm, we must model the middleware as any other component. Moreover, the communication between components, thus between the middleware and the other components, is fixed: It depends on how automata are rendered in the analysis language. For instance, vUML would not allow us to model a middleware which guarantees that the order of publication is kept also during notification. These approaches also impose that channels between components be explicitly declared: vUML and veriUML do not allow us to create or destroy components at run-time and the topology of the communication is fixed.

The proposals presented so far do not support a friendly language to define properties. With vUML we can only state reachability properties, while with veriUML, JACK, and HUGO we can also define complex properties on how the application evolves, but in all cases they must be declared directly in the formalism supported by the model checker, that is, CTL, ACTL and LTL, respectively. All these formalisms are based on temporal logic and are difficult to use and understand by designers with no specific background.

Two other projects try to overcome these limitations (i.e., definition of properties and communication paradigm). Inverardi et al. [11] apply model checking techniques to automata that communicate through channels. In this approach, properties are specified graphically through MSCs. They support two kinds of properties: (a) the application behaves at least once as the MSC, or (b) the application must be always complaint with the MSC. MSCs are directly translated into LTL, the property language supported by SPIN.

Kaveh and Emmerich [12] exploit model checking techniques to verify distributed applications based on remote method invocation. Components are described through statechart diagrams where if a transition fires, some remote methods are invoked. Only potential deadlocks can be discovered by this tool.

Garlan et al. [5] and the Cadena project [8] apply model checking techniques on distributed publish/subscribe architectures. Both of these proposals do not deal with UML diagrams, but define the behavior of components through a proprietary language [5] or using the CORBA IDL-like specification language [8].

Garlan et al. provide different middleware specifications that can be integrated into the validation tool. The properties are specified in CTL, which is the formalism provided by the SMV [15] model checker. Although the ideas in this

proposal and in our approach are similar, there are some differences: (a) we provide a complete graphical front-end for the designer that does not have to deal with any particular textual and logical formalism, (b) the set of warranties supported by our middleware is wider (e.g., [5] does not deal with subscriptions), and (c) LSCs provide the operators for describing the communication among components in a graphical and natural way, while CTL is a general-purpose temporal logic.

Cadena, which is the other proposal that supports publish/subscribe architectures, only deals with the CORBA Component Model (CCM). In Cadena, the communication is established explicitly, that is, each component declares the components from which it desires to receive events. This particular implementation of the publish/subscribe paradigm does not allow the use of Cadena with other middleware platforms. Cadena supports the Bandera Specification Language [3] to specify properties against which the system must be validated.

## 5. CONCLUSIONS AND FUTURE WORK

In this paper we present an approach to model and validate distributed architectures based on the publish/subscribe paradigm. Application-specific components are modeled as UML statechart diagrams while the middleware is supplied as a configurable predefined component. As to validation, properties are described with *live sequence charts* (LSCs) and transformed into automata. Components, middleware, and properties are translated into Promela and then passed to SPIN to validate the architecture.

Our future work is headed to different directions. We are extending the approach to model time and probabilities associated with publication/notification of events. But we are also trying to understand how these analyses can be performed in an incremental way: do properties remain valid? What about results?

Finally, we are studying how to better support the designer while modeling applications, the adoption of different model checkers to understand the impact they have on obtained results, and the possibility of automatically coding the infrastructure of these architectures by means of the analysis models.

## 6. REFERENCES


[1] A. Carzaniga and D. S. Rosenblum and A. L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332–383, Aug 2001.

[2] K. Compton, Y. Gurevich, J. Huggins, and W. Shen. An automatic verification tool for UML. Technical Report CSE-TR-423-00, 2000.

[3] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. A language framework for expressing checkable properties of dynamic software. In *Proceedings of the SPIN Software Model Checking Workshop*, volume 1885 of *LNCS*, August 2000.

[4] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transaction on Software Engineerings*, 27(9):827–850, September 2001.

[5] D. Garlan and S.Khersonsky and J.S. Kim. Model Checking Publish-Subscribe Systems. In *Proceedings of the 10th SPIN Workshop*, volume 2648 of *LNCS*, May 2003.

[6] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[7] S. Gnesi, D. Latella, and M. Massink. Model Checking UML Statecharts Diagrams using JACK. In *Proc. Fourth IEEE International Symposium on High Assuarance Systems Enginering (HASE)*, pages 46–55. IEEE Press, 1999.

[8] J. Hatcliff, W. Deng, M.D. Dwyer, G. Jung, and V. Ranganath. Cadena: An integrated development, analysys, and verification environment for component-based systems. To appear in Proc. of the International Conference on Software Engineering (ICSE 2003),IEEE Press,2003.

[9] G.J. Holzmann. *Design and Validation of Network Protocols*. Prentice Hall, 1991.

[10] G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[11] P. Inverardi, H. Muccini, and P. Pelliccione. Automated check of architectural models consistency using SPIN. In *Proc. Automated Software Engineering conference (ASE2001)*, pages 349–349. IEEE Press, 2001.

[12] N. Kaveh and W. Emmerich. Deadlock detection in distributed object systems. In *Proc. of the Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9)*, pages 44–51, Vienna, Austria, 2001. ACM Press.

[13] L. Zanolin and C. Ghezzi and L. Baresi. Model Checking Pusblish/Subscribe Architectures. Technical report, 2003.

[14] J. Lilius and I.P. Paltor. vUML: a tool for verifying UML models. In *Proc. 14th IEEE International Conference on Automated Software Engineering (ASE)*, pages 255–258, Cocoa Beach, Florida, October 1999.

[15] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publ., 1993.

[16] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. Foundations of Computer Science(FOCS-77)*, pages 46–57, Providence, Rhode Island, October 1977.

[17] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley Lognman, 1999.

[18] T. Schäfer, A. Knapp, and S. Merz. Model checking UML state machines and collaborations. *Electronic Notes in Theoretical Computer Science*, 55(3):13 pages, 2001.

[19] Sun Microsystem. Java Message Service Specification. Technical report, Sun Microsystem Technical Report.

[20] TIBOC. The Power of now. Tibco hawk. http://www.tibco.com/solutions/.

# Timed Probabilistic Reasoning on UML Specialization for Fault Tolerant Component Based Architectures

Jane Jayaputera, Iman Poernomo and Heinz Schmidt
{janej,ihp,hws}@csse.monash.edu.au
CSSE, Monash University, Australia

## ABSTRACT

Architecture-based reasoning about reliability and fault tolerance is gaining increasing importance as component-based software architectures become more widespread. Architectural description languages (ADLs) are used to specify high-level views of software design. ADLs usually involve a static, structural view of a system together with a dynamic, state-transition-style semantics, facilitating specification and analysis of distributed and event-based systems. The aim is a compositional syntax and semantics: overall component behavior is understood in terms of subcomponent behavior. ADLs have been successful in understanding architecture functionality. However, it remains to be investigated how to equip an ADL with a compositional semantics for specification and analysis of extra-functional properties such as reliability and fault-tolerance.

This paper combines architecture definition with probabilistic finite state machines suitable to model reliability and fault-tolerance aspects. We present a compositional approach to specifying fault tolerance through parameterization of architectures. Using Probabilistic Real Time Computational Tree Logic (PCTL) we can specify and check statements about reliability of such architectures.

## 1. INTRODUCTION

In distributed systems, fault-tolerance can be aided by replication mechanisms. Central to these mechanisms is the notion of *fail-over*: a backup server takes over the job from a crashed server after a short timeout period and sends data back to the client directly, without human reconfiguration, as if the original server is still operating. There is a range of possible replication algorithms for achieving fail-over. We would like to systematically apply similar kinds of fail-over to design more reliable component-based software architectures. Given particular client and server components, we wish to create a fault tolerant architecture by associating a replication algorithm with calls to the server from the client.

Architectural description languages (ADLs) are used to specify high-level views of software design. ADLs usually involve a static, structural view of a system with a dynamic, state-transition style semantics, facilitating specification and analysis of distributed and event-based systems. The implementation is compositional: component behavior is understood in terms of subcomponent behavior.

However, the compositionality of component specifications cannot be taken for granted, especially when extra functional properties of the dynamic behavior are modeled [18]. In previous work [14] we have developed a compositional ADL-based approach to reliability using Markov chains.

In this paper, we extend some of those ideas, focusing on representing a range of fail-over replication strategies in the syntax and semantics of a compositional ADL. The novelty of our approach is the combined use of

- parameterization of architectures to treat replication strategies systematically, and
- a probabilistic semantics to facilitate fault-tolerance analysis of resulting architectures.

We can then use our work to reason about reliability properties of software component architectures. Our approach is simple, combining three formalisms:

1. We define an ADL with probabilistic finite state machine (PFSM) semantics. The semantics tells us about the dynamic behavior of a component. Specifically, it permits us to define how a call to a component interface method will result in calls to other required components. Our semantics is probabilistic, so it permits our models to relate usage profiles of method calls (the probability that particular sequences of methods will be called), and also to model method reliability (i.e., the probability of method execution success). Overall component reliability is then given as a cumulative function of method reliability over all component interfaces.
2. We apply a parameterization mechanism to add fault tolerant features automatically into the ADL. The parameterization involves choosing one of the provided replication algorithms. These algorithms replicate the server process to enable fail-over.
3. We use Probabilistic Real Time Computational Tree Logic (PCTL) to specify and check statements about such architectures. This is possible because PCTL statements have truth values that are determined according to the ADLs probabilistic finite state machine semantics, if we associate logical properties with particular states. To check statements against architectures, we use the compositional semantics to build a machine for the architecture, preserving the logical properties known to hold for subcomponents. This yields a larger PFSM over which PCTL statements may be checked.

## 2. FAULT TOLERANCE IN DISTRIBUTED SYSTEMS

Reliability and fault-tolerance are some of the key issues of current distributed systems. Large scale, widely distributed systems contain large numbers of network nodes and connections. There is a likelihood that some nodes or some connections will be unavailable. Because many connections and

intermediate nodes are needed to enable a client-server communication, this temporary unavailability can significantly decrease the overall reliability of a system. This is of particular concern in distributed enterprise systems.

Replication combined with a fail-over logic is a common fault tolerance mechanism, overcoming some of these problems and hence increasing overall availability and reliability.

Replication mechanisms try to overcome system failure by duplicating processes and resources. Then clients can access resources without having to worry about server crashes and unpredicted downtime. This is possible because requests made to a crashed server are now diverted to another server (or *replica*). The replica then sends information back to clients directly, as if the original server itself was performing the data transfer.

A range of fail-over replication algorithms has been proposed. *Passive replication* (also known as primary-backup) [2] and *active replication* (also known as state machine) [20] approaches are probably the most important and well-used ones. Most of the remaining approaches are extension of these two replication methods. There are several important extensions. *Active client in passive replication* approach, abbreviated here as *active client replication*) [3], extends the primary-backup approach by making a client actively choose a server to contact. Recently, [4, 17] specified a new replication technique known as *semi-passive replication* mechanism. For reasons of scope, we only describe passive replication and active client replication algorithms in this paper. Interested parties are referred to, for example, [7] for details on the other algorithms.

The passive replication approach mainly works as follows. At any one time, there is at most one primary server to serve requests from clients. Other servers act as backups. These backups receive the updated data from the primary server and do not interact directly with clients. If the primary server fails, one of the backups takes over the serving role and acts as the new primary server [2].

In order to overcome the main drawback of passive replication approach, active client replication was invented. It extended the passive replication algorithm by allowing client to contact a backup server directly if the primary server does not function correctly (crash) [3]. One of the current authors has extended the approach by allowing busy server to be handled as well as crashed primary server and message omission failure [7].

Our probabilistic fail-over model uses the following probabilities (illustrated in Fig. 1).

- $P_{primary}(S)$ is the probability of server $S$ to be chosen as primary server when a client makes a request.
- $P_{busy}(S)$ is the average probability of server $S$ being busy for a call at any time.
- $P_{bottleneck}(S)$ is the failure probability of server $S$.
- $P_{transfer}(S, S')$ gives the probability of server $S'$ being chosen as backup server in lieu of server $S$.

Without loss of generality, for simplicity of the examples, we assume that these probabilities are independent of $S$.

## 3. ARCHITECTURAL DESCRIPTION LANGUAGE

Architectural Description Languages (ADLs) are used to specify coarse-grain components and their overall interconnection structure. ADLs are compositional, permitting the

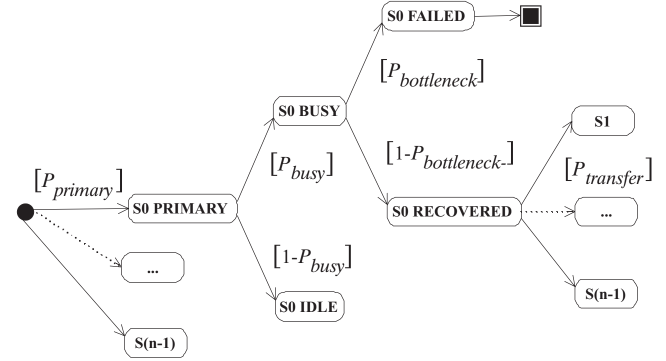



*Figure 1: Probabilities.*

specification of components in terms of smaller components. Examples of ADLs are Darwin [10], Wright [1] and radl [19].

In this section, we define a simplified version of radl, a language for describing and analyzing both functional and nonfunctional properties of architectures. Like many ADLs, radl consists of a visual and textual notation for defining the static composition of a system, and a state transition semantics for analysis of dynamic aspects. We describe the former and then the latter.

The basic elements of our language are components – referred to as *kens* in radl.[1] The functionality of a component is defined by a set of provided and required ports. The ports are referred to as *gates* in radl. Gates are to be regarded as interfaces of the component, which can be accessed by an external client. The internal functions of a ken are specified by internal ports, which can be thought of as internal methods used to implement an interface, not available to an external client. Provided and required gates express the external functionality that a ken provides and needs to use, respectively.

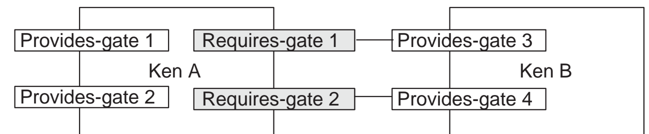



*Figure 2: Example of radl.*

## 4. FAULT TOLERANCE PARAMETERIZATION IN ADL

Replication is usually described as communications between a client with some servers in distributed systems. We propose a method to abstract over these communications in component-based software architecture. We encapsulate the detailed design of server replications using an abstraction concept, added into the ADL. Instead of expecting a software designer to know the details of a particular algorithm, we provide a mechanism to automatically produce a fault tolerant architecture, given client and server components and a chosen algorithm. The mechanism is called *Parameterized Replication*.

[1] We use the term *ken* to specify the elements of our ADL, as these elements are often more general than traditional system components, representing a range of other architectural building blocks, such as transactional boundaries or, in our case, parameterized fault-tolerant architectures.

We allow kens to be parameterized by different replication strategies, through use of the *ParRepl* construct. Like UML template classes (and template components), we draw a parameter as a box in the corner of a ken to illustrate the parameterized replication notation.

The first abstraction represents a black-box of parameterized replication. The parameterization shows the name of the replication algorithm that is used (see for example Fig. 3). In this abstraction, an architect does not have to know the algorithm details. The architect need only specify a name of replication algorithm that will be used (in this case AR stands for active replication).

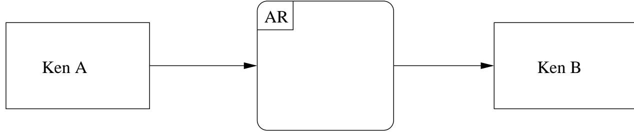



*Figure 3: Abstraction 1.*

In radl, the first abstraction corresponds to a new construct

$$\texttt{ParRepl}(C, S, algo, probs, n)$$

This element wraps all interactions between a client $C$ and server $S$ according to a replication algorithm defined by *algo*. The algorithm ranges over fail-over mechanisms, replicating $n$ copies of the server $S$ by $n$. We assume that $\{S_0, S_1, ..., S_{n-1}\}$ are a set of servers. We denote the algorithms mentioned above by `PassiveReplAlgo`, `ActiveReplAlgo`, `SemiPassiveReplAlgo`, `ActiveClientReplAlgo` with their obvious meaning. We use probabilities *probs* specified at the end of section 2 for reliability measurement of fault tolerant architecture.

Fig. 2 presents two basic kens *Ken A* and *Ken B* with two bindings between required and provided gates (shaded and white rectangles, respectively). *Ken A* is a client of *Ken B*, a server. Fig. 4 shows a variation of Fig 2. *Ken B* has been parameterized by the formal parameter of the replication algorithm. A particular replication algorithm, Passive Replication, has been chosen as an actual parameter.

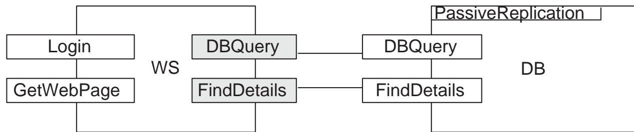



*Figure 4: Example of syntax for fault tolerant radl.*

The second abstraction is concerned with the actual communication of client and servers according to a particular algorithm. The active replication algorithm is shown in Fig. 5 (a), where a client communicates with all servers. Fig. 5 (b) shows passive replication algorithm, where a client can contact a primary server only. This abstraction is elaborated in section 5.

## 5. SEMANTICS FOR DYNAMIC ANALYSIS

Architectures of radl are equipped with probabilistic finite state machine (PFSM) semantics. Our semantics is compositional, in the following sense. Each basic ken is associated with a set of PFSMs, defining how calls to provided gates

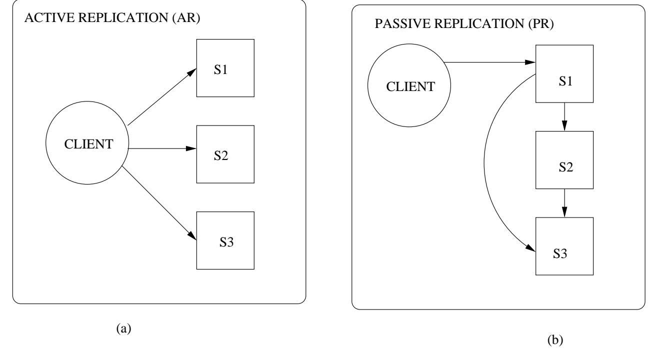



*Figure 5: Abstraction 2.*

result in internal gate calls and outgoing signals to required gates. Then, our semantics defines how larger compositions of kens result in larger PFSMs from the individual kens' PFSMs.

A PFSM may be formally defined:

Definition 5.1 (PFSM). *A Probabilistic Finite State Machine (PFSM) is a tuple*

$$D = \langle E_D, A_D, Z_D, \delta_D, Prob_D, initial_D, Final_D, failed_D \rangle$$

*$E_D$ is called event alphabet, $A_D$ is the action alphabet, $Z_D$ denotes the set of states. $initial_D, failed_D \in Z_D$ are designated initial and fail state, respectively and $Final_D \subset Z_D$ is the set of final states.*

$$\delta_D : Z_D \times E_D \cup A_D \rightarrow Z_D$$

*and*

$$Prob_D : Z_D \times E_D \cup A_D \rightarrow [0, 1]$$

*are the transition function and the transition probability, respectively, where for all $z \in Z_D$:*

$$1 = \sum_{x \in E_D \cup A_D} Prob_D(z, x)$$

*In other words $Prob_D$ induces a probability distribution on the set of transitions for each state.*

For the sake of conformance to standards for interchangeability within CASE tools, our visual notation for PFSMs borrows from UML statechart diagrams and extends them with probabilities. Initial, failed and final states are marked using a solid circle, a solid rectangle and a circle marked with a dot, respectively. In our extension, a transition from state $s_1$ to $s_2$ is of the form:

$$s_1 \xrightarrow{e[p]/a} s_2$$

where

- $e$ is an optional *event*. We use events to denote calls to a ken's provided or internal gates, resulting in the state transition
- $a$ is an optional *action*. We use actions to denote internal activity or calls to a ken's required gates.
- $p$ is a probability value (the probability of the given transition). ¿From a practical viewpoint, $p$ is often the product $u \cdot r$ where $u$ is the usage probability (of

the event $e$) and $r$ the reliability of the action $a$. For details see [14].

A PFSM $D$ then gives rise to a (finite-state discrete-time) Markov chain (transition probability matrix) $M$ by summing the (multi-edge) probabilities between a given pair of states and solving the resulting model analytically [14].

### Composing Fault Tolerant PFSM

Our ADL has been defined as a Meta Object Facility (MOF) meta-model through UML specialization [12]. The details are topic for another paper. Here it suffices to mention that the MOF/UML permits to treat architectural and behavioral considerations in tandem. Kens are treated as metaclasses that contain a meta-method *semantics*() which returns a PFSM model (as a first class object) accessible to other methods in our implementation.

We define a fault tolerant parameterization semantics as follows:

1. There is a UML meta-model for describing our ADL and its semantics, mainly defined by `StateMachine`, `Transition`, and `State` [11]. They are similar to `PFSM`, $\delta_D$ and $Z_D$ in Def. 5.1, respectively.

2. The replication algorithms are given by a metaclass ReplAlgo, used to generically compute semantics for a given replication choice. This metaclass is equipped with a virtual (meta-)method *execAlgo* that denotes the algorithm. The range of replication algorithms may then be represented by subclassing ReplAlgo and redefining this function.

   Some algorithms inherit properties of ReplAlgo abstract class: *PassiveReplAlgo*, *ActiveReplAlgo*, *SemiPassiveReplAlgo*, and *ActiveClientReplAlgo*. This approach is efficent from a meta-modeling perspective, because it enables us to treat all algorithms as of the metaclass type ReplAlgo.

3. A *ParRepl* is given as a parameterized (meta-)class that takes four arguments. Given client PFSM and server PFSM, it calls an abstract class called *ReplAlgo* and outputs a combined PFSM as the result.

Another novelty of this approach is that we use MOF and UML meta-modeling to generically treat architectures over particular replication algorithms.

We outline a compositional semantics for fault tolerant architectures, based upon PFSMs. We refer the interested reader to [13] for a detailed formal description of a similar semantics using deterministic finite state machines without probabilities and [8] with probabilities. We define passive replication algorithm in this paper to illustrate composition of fault tolerant architectures.

**Binding.** Given ken $C$ with required gate $pp$ connected to ken $S$ with provided gate $pp$,

$$Bind(C, S, pp)$$

we build a larger set of PFSMs associated with the provided gates of $Bind(C, S, pp)$

$$[\![Bind(C, S, pp)]\!]$$

according to the following algorithm. First, let

$$Sem_0 = [\![S]\!] - \{PFSM_{pp}\}$$

1. Set $i := 0$.

2. For each

   $$PFSM_{epp} \in [\![C]\!]$$

3. If $PFSM_{epp}$ does not involve $pp$ as an action, then let $Sem_{i+1} = Sem_i \cup \{PFSM_{epp}\}$, let $i := i + 1$, take the next $PFSM_{epp} \in [\![C]\!]$ and go to step 3.

   Otherwise, go to step 4.

4. Take every transition in $PFSM_{epp}$ that involves $pp$ as an action,

   $$s_1 \overset{e[call*rel]/pp}{\longrightarrow} s_2 \quad (1)$$

   and let $RPFSM_{pp}$ be the $PFSM$ for the provided gate $pp$ in $[\![S]\!]$. We define $Change(RPFSM_{pp}, s_2)$ to be $RPFSM_{pp}$ with its *initial* state replaced by $s_2.pp.start$ and $final$ state with $s_2.pp.end$. Then, we insert $Change(RPFSM_{pp}, s_2)$ between $s_1$ and $s_2$, in the sense that we delete the transition 1 and replace it with

   $$s_1 \overset{e[call*rel]/pp}{\longrightarrow} s_2.pp.start$$

   and add the transition

   $$s_2.pp.end \overset{pp.end[1]}{\longrightarrow} s_2$$

   We call the resulting machine $PFSM'_{epp}$.

5. We let $Sem_{i+1} = Sem_i \cup \{PFSM'_{epp}\}$, take the next $PFSM_{epp} \in [\![C]\!]$, set $i := i + 1$ and go to step 3.

**ParRepl.** We define the semantics of the fault-tolerant replication

$$[\![ParRepl(C, S, algo, probs, numServers)]\!]$$

according to the algorithm defined in $algo$. Here we only give a sketch of the semantics. The probabilities $primary(S)$ are added in the transition between client PFSM to the PFSM for each servers $S$, by defining a new state $S.current$ which intercedes calls from the client with $primary(S)$ as the probability of transition from the client call. Probability $busy(S)$ is added in the transition after $S.current$, by defining transition $busy(S)$ to a new state $S.BUSY$. $idle(S)$ is the probability of server $S$ being idle (does not fail). The probability $bottleneck(S)$ is added in the transition between $S.BUSY$ to the root PFSM. Probabilities $transfer(S, S')$ are added in the transition between $S1.transferControl$ to other servers' PFSM starting state $S'.current$. Probability $otherFailures(S)$ is being added in the transition between $S.BUSY$ to a new state $S.transferControl$.

**ReplAlgo.** As an illustration, we define the semantics for $ParRepl$ where $algo$ is set to `PassiveReplAlgo`.

1. Set n := 0 and do all steps in $Bind(C, S, pp)$ up to step 3. Instead of calling step 4, refer the call to this algorithm in step 2.

   Let us add some probabilistic transitions to each of ken $S$'s $PFSM$s in steps 3 for correct processes and 4 for busy processes.

2. For each $PFSM_n$
   s.t. $n < numServers$ and $PFSM_{pp} \in [\![S]\!]$:

3. Go to step 4 in $Bind(C, S, pp)$ to compose a larger PFSM in normal case (no failure). After that, we do a composition for failure cases. We define a new function $Add(FTPFSM_{pp}, s_1, s_2)$ to add a new state, namely $s_2.pp.current$, so that we can put a probability in choosing a primary server $probs[n][primary]$ for all $numServers$ servers. Then, we insert the function $Add(FTPFSM_{pp}, s_1, s_2)$ so that two more transitions are added between $s_1$ and $s_2$ (besides the transitions in step 4 of $Bind(C, S, pp)$) with

$$s_1 \overset{probs[n][primary]}{\longrightarrow} s_2.pp.current$$

and

$$s_2.pp.current \overset{probs[n][idle]}{\longrightarrow} s_2.pp.start$$

4. Add a transition to a new state $S_n.BUSY$ if busy processes happen:

$$s_2.pp.current \overset{probs[n][busy]}{\longrightarrow} S_n.BUSY$$

In the case of bottleneck (crashes) without having a chance to transfer the control to a new primary:

$$S_n.BUSY \overset{probs[n][bottleneck]}{\longrightarrow} final$$

Add a transition in the case of failures other than bottleneck:

$$S_n.BUSY \overset{probs[n][other]}{\longrightarrow} S_n.transferControl$$

And add other transitions to the starting state of other backup servers:
For $((0 \le backup < numServers) \wedge (backup \ne n))$:

$$S_n.transferControl \overset{probs[n][transfer][backup]}{\longrightarrow} s_2.pp.current$$

Then set $n := n + 1$ and repeat step 3 until $n = numServers$.

5. We call the resulting machine $PFSM'_{epp}$ and go to step 5 in $Bind(C, S, pp)$.

## 6. PROBABILISTIC COMPUTATIONAL TREE LOGIC

Probabilistic Computational Tree Logic (PCTL) (see for instance [6]) is used to specify and check timing and probabilistic properties on our architectures. We define PCTL in terms of *structures* comprising PFSMs and associating additional atomic propositions with states:

Definition 6.1 (Structure). *A* structure *is a tuple*

$$S = \langle D_S, Prop_S, h_S \rangle$$

*where $D_S$ is a PFSM, $Prop_S$ is a finite set of atomic propositions and $h : Z_{D_S} \to \mathcal{P}(Prop_S)$ is a function decorating states with propositions sets.*

The idea is to extend each basic ken PFSM by associating atomic propositions with states. We use the architectural semantics of the previous section, so that compositions of kens have appropriately expanded structures. Formulae can then be specified about the provided gates of any given architectural composition, and then checked against the associated structures.

We extend the traditional definition of PCTL which uses transition probabilities without associated events or actions. To this end we use finite sequences in $(E \cup A)^*$, infinite sequence in $(E \cup A)^\omega$ and bounded sequences in $(E \cup A)^{n<k}$ for some bound $k$. Such a sequence $a_0 \dots a_{n-1}$ is also called a finite, infinite or bounded *trace*, respectively, of the PFSM $D$, if there is an associated sequence of states $s_0, \dots, s_{n-1} \in Z_D$ such that $initial_D = s_0 \overset{a_0}{\to} s_1 \overset{a_1}{\to} \cdots s_{n-1}$ are legitimate transitions, i.e., if $\delta_D(s_i, a_i) = s_{i+1}$ for all for the respective step $i$. The probability of this trace is the product of the single transition probabilities: $p(a_0 \dots a_{n-1}) = \prod_i Prob_D(s_i, a_i)$. The trace is also called *accepted* if $s_{n-1} \in Final_D$.

As computation tree paths in PCTL are sequences of states without transition symbols, in order to forget the transition symbols we map a given trace $t = a_0 \dots a_{n-1}$ to its underlying state sequence $s = s_0 \dots s_{n-1}$. We denote by $s_t$ the underlying state sequence of $t$. Since different symbols can make different transitions between the same pair of states, then different traces can have the same underlying state sequence. By $T_s = \{t \mid s_t = s\}$ we denote the set of traces with the same underlying state sequence $s$.

Now the paths in the sense of traditional PCTL are the underlying state sequences of the traces in $T_D$. Their probability can be computed as

$$p(s) = \sum_{t \in T_s} p(t)$$

This means we sum the trace probabilities given in the PFSM over all traces with the same underlying state sequence to derive the probability of the state sequence.

PCTL now permits model checking with temporal formulae that are composed from atomic propositions and include modal operators with optional lower reliability bounds and upper time bounds. Regarding the time bounds, such operators are interpreted over all traces (and state sequences) up to the given length in a time bound. Bounded reliability is defined using the probabilities above, where we sum the probabilities over all traces (implicitly over all underlying state sequences) satisfying the given formulae, i.e. over a corresponding state 'computation tree'.

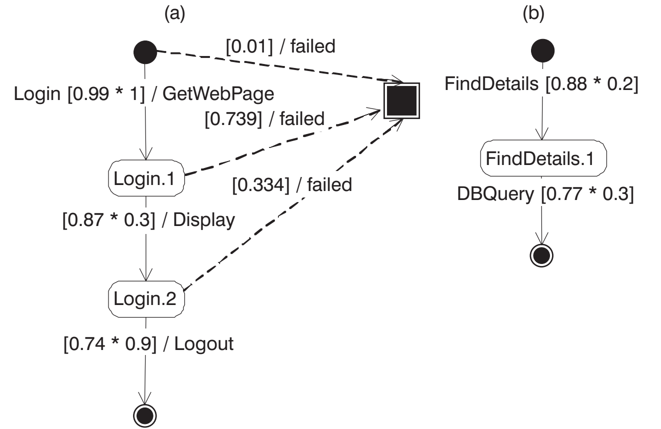



*Figure 6: PFSMs associated with basic kens of our example: (a) $PFSM_{\texttt{Login}} \in [\![\texttt{WS}]\!]$, (b) $PFSM_{\texttt{FindDetails}} \in [\![\texttt{DB}]\!]$, and The failed state and its incoming transitions are displayed in chart (a) and are left implicit in the other chart.*

Without loss of generality and for the sake of simplicity in the following examples, we now assume that the above probabilities are all independent of the chosen server(s). In

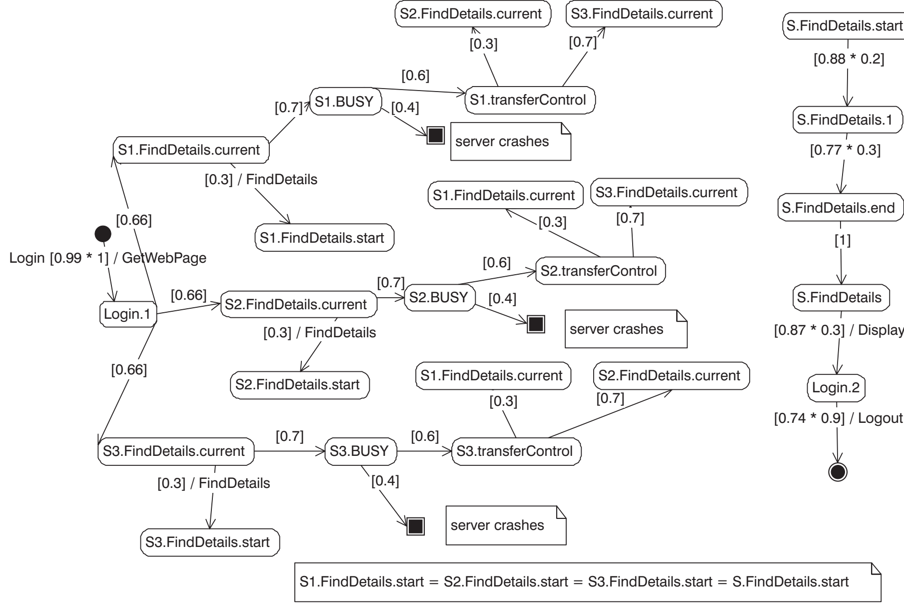



*Figure 7: Fault tolerant PFSM using passive replication.*

other words, we assume a homogeneous pool of servers with symmetric fail-over policy.

EXAMPLE 6.1. *Using* `PassiveReplAlgo`*, let $NewDBReq$ be an atomic proposition, standing for the fact that a new database connection that fault tolerant needs to be added to the pool in our example. Assume that $h_{\mathtt{DB}}(FindDetails.1) = h_{\mathtt{WS}}(Login.2) = \{NewDBReq\}$, so that*

$$h_{\mathtt{ARCH}}(S1.FindDetails.1) = h_{\mathtt{ARCH}}(C.Login.2) = \{NewDBReq\}$$

*We make the following specification about the behavior of a call to the* `Login` *gate of* `ARCH`*. In the best case, there is a probability of at least 0.2% that we will require a new database connection in less than or equal to 7 times steps in server $S_1$.*

$$True\ U^{t\leq 7}_{p\geq 0.002}\ NewDBReq \tag{2}$$

*After the third attempt, we get a probability of at least 0.37% that we will require a new database connection in less than or equal to 14 times steps in server $S_3$, using the same formulae above.*

*For the overall architecture, we get a reliability of at least 82% if we use one server only. The reliability is increased by 16.3% if we use three servers instead.*

$$True\ U^{t\leq 8}_{p\geq 0.82}\ final \tag{3}$$

*Using the algorithms of [6], we can verify that this specification is true of* `ARCH`*.*

EXAMPLE 6.2. *Assuming active client replication algorithm is used, we use the same atomic proposition as in Example 6.1. For the best case, we have probability of at least 0.2% that we will require a new database connection in less than 7 times steps in server $S_1$ using Equation (2). After the third attempt, we get a probability of at least 0.38% that we will require a new database connection in less than or equal to 17 times steps in server $S_3$, using the same formulae.*

*For the overall architecture, at least 60% reliability is achieved if we use one server only. By replicating the number of servers to three, we get a greater reliability of at least 98.2%. We use the formulae in Equation (3) for both measurements.*

The PFSM in Fig. 7 is the result of combining individual PFSMs in Fig. 6, using the composition algorithm for passive replication defined in Section 5.

In relation to composing PFSMs using the active client approach in Example 6.2, more states need to be used compared to Example 6.1. This is due to the need to choose a new primary server in case the old primary fails. Thus, the client has to resend a request to the new primary and the corresponding client PFSM has to be included again in the composition. As the number of states increases, the number of steps also increases (21 steps in Example 6.1 compared to 30 in Example 6.2).

Updating backup servers (as part of the passive replication algorithm) is not included in the PFSM composition. We can avoid this because our reliability measurements are only the sequences leading to the client receiving a response regardless of server replication. It can be seen from Example 6.1 and Example 6.2 that reliability of architecture is increased by around 0.01% if we use three servers instead of one server only. Since a crashed server can often be restarted after a failure, there is an additional element of availability 'built-in' that we are not even accounting for, since our simplified model treats failure as terminal.

## 7. IMPLEMENTATION

The compositional semantics for our version of radl have been implemented. The software, called *FSMComb* [5], implements our methods in Java. FSMComb interfaces to the PRISM [9] model checking tool which permits to check PCTL specifications against our architectures. Generally, FSMComb combines two or more PFSMs for individual kens into one for the given architecture. The PFSMs are read from text files and then extended by fault-tolerance constructs as described. Finally the composite models are exported to the PRISM checker together with fault-tolerance assertions. PRISM then verifies these models and reports the results.

## 8. RELATED WORK AND CONCLUSIONS

Little work has been done in using ADLs to specify and analyze fault tolerant properties.

In [16], non-functional properties such as dependability, safety, reliability and availability are defined formally using a predicate logic with some extensions. Then, components and replication methods are also defined formally using the non-functional properties that have formally been defined previously. However, that work does not relate a compositional ADL-style view of architecture to replication techniques. The relation of fault tolerance to executable architectures was investigated in [15]. The approach adds fault tolerant supports into SOFA component framework. The SOFA framework is based on component oriented programming. Thus, SOFA is similar to OMG's CORBA, Sun's Enterprise Java Beans and Microsoft's COM. Although they use replication methods such as active and passive replication as our approach, there are some differences.

The approach does not replicate a component automatically. Also, the primary goal of that work is to implement fail-over algorithms directly in SOFA source code, without describing it abstractly in architecture.

In [14], we developed a compositional approach to reliability models where PFSMs are associated to hierarchical component definitions and to connectors. Markov chain semantics permits hierarchical composition of these reliability models. However the paper does not develop a fault-tolerance model. The work presented here is in part based on that semantics.

This paper presented a compositional approach to fault-tolerant component-based architectures. We modeled fail-over mechanisms in a pool of replicated servers. PFSMs were associated with components and connectors to define the behavior of hierarchical component-based architectures. We sketched a formal semantics using PCTL.

Parameterized architectural patterns are used in our approach, in which the chosen fault-tolerance mechanisms becomes a parameter. The actualization of the parameter includes probabilities for weaving the PCTL fault-tolerance model into the PCTL models of the client-server interface functionality.

Finally, the paper developed an example for the special case of a pool of symmetric servers - although our approach is more general. The example also illustrated our prototype implementation of the PCTL architecture weaver.

## 9. REFERENCES


[1] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, May 1997.

[2] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. In S. Mullender, editor, *Distributed Systems*, chapter 8, pages 199–215. ACM Press, 2nd edition, 1993.

[3] P. Chundi. *Protocols for Achieving Consistency and Reliability in Replicated Database Systems that Utilize Asynchronous Updates*. PhD thesis, Department of Computer Science, University of Albany - State University of New York, August 1996.

[4] X. Défago and A. Schiper. Specification of replication techniques, semi-passive replication and lazy consensus. Technical Report IC/2002/007, École Polytechnique Fédérale de Lausanne, Switzerland, Feb. 2002.

[5] Fsmcomb - finite state machine combinator. See `http://www.csse.monash.edu.au/~janej`.

[6] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.

[7] J. Jayaputera. *Fault Tolerance in Active Client/Passive Replication: Tolerating Faulty or Slow Servers and Handling Network Partitions*. Honours Thesis, School of CSIT, RMIT University, Oct 2001.

[8] J. Jayaputera, I. Poernomo, R. Reussner, and H. Schmidt. Timed probabilistic reasoning on component based architectures. In H. Sondergaad, editor, *Third Australian Workshop on Computational Logic*. ANU, Canberra, Dec 2002.

[9] M. Kwiatkowska, G. Norman, and D. Parker. Prism: Probabilistic symbolic model checker. In *PAPM/PROBMIV'01 Tools Session*, 2001.

[10] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In W. Schafer and P. Botella, editors, *Proc. 5th European Software Engineering Conf.*, volume 989, pages 137–153, Sitges, Spain, 1995. Springer-Verlag, Berlin.

[11] OMG. *UML Specification v1.5*, March 2003.

[12] OMG. *UML Superstructure v2.0*, April 2003.

[13] R. Reussner, I. Poernomo, and H. Schmidt. Using the TrustME Tool Suite for Automatic Component Protocol Adaptation. In P. Sloot, J. Dongarra, and C. Tan, editors, *Computational Science, ICCS 2002, The Netherlands, 2002*, volume 2330 of *LNCS*, pages 854–862. Springer-Verlag, Berlin, Germany, Apr. 2002.

[14] R. Reussner, H. Schmidt, and I. Poernomo. Reliability prediction for component-based software architectures. *Journal of Systems and Software – Special Issue of Software Architecture - Engineering Quality Attributes*, 66(3):241–252, 2003.

[15] J. Rovner. Fault tolerant support for sofa. Technical report, Dept. of CSE, University of West Bohemia in Pilsen, Czech Republic, 2001.

[16] T. Saridakis and V. Issarny. Developing dependable systems using software architecture. In *Proceedings of the 1st Working IFIP Conference on Software Architecture*, pages 83–104, San Antonio, TX, USA, Feb 1999.

[17] A. Schiper. Failure detection vs. group membership in fault-tolerant distributed systems: Hidden trade-offs. In *PAPM-PROBMIV 2002*, LNCS 2399, pages 1–15, Denmark, July 2002. Springer Verlag. Invited talk.

[18] H. Schmidt. Trustworthy components – compositionality and prediction. *Journal of Systems and Software – Special Issue of Software Architecture - Engineering Quality Attributes*, 65(3):215–225, 2003.

[19] H. Schmidt, I. Poernomo, and R. Reussner. Trust-By-Contract: Modelling, Analysing and Predicting Behaviour in Software Architectures. In *Journal of Integrated Design and Process Science*, volume 4(3), pages 25–51, 2001.

[20] F. B. Schneider. Replication management using the state-machine approach. In S. Mullender, editor, *Distributed Systems*, chapter 7, pages 169–197. ACM Press, 2nd edition, 1993.

# Modelling a Framework for Plugins

Robert Chatley
Department of Computing
Imperial College London
180 Queen's Gate, London
SW7 2AZ
rbc@doc.ic.ac.uk

Susan Eisenbach
Department of Computing
Imperial College London
180 Queen's Gate, London
SW7 2AZ
sue@doc.ic.ac.uk

Jeff Magee
Department of Computing
Imperial College London
180 Queen's Gate, London
SW7 2AZ
jnm@doc.ic.ac.uk

## ABSTRACT

Using plugins as a mechanism for extending applications to provide extra functionality is appealing, but current implementations are limited in scope. We have designed a framework to allow the construction of flexible and complex systems from plugin components. In this paper we describe how the use of modelling techniques helped in the exploration of design issues and refine our ideas before implementing them. We present both an informal model and a formal specification produced using Alloy. Alloy's associated tools allowed us to analyse the plugin system's behaviour statically.

## Keywords

plugins, components, modelling, specification

## 1. INTRODUCTION

Maintenance is a very important part of the software development process. Almost all software will need to go through some form of evolution over the course of its lifetime to keep pace with changes in requirements and to fix bugs and problems with the software as they are discovered.

Traditionally, performing upgrades, fixes or reconfigurations on a software system has required either recompilation of the source code or at least stopping and restarting the system. High availability and safety critical systems have high costs and risks associated with shutting them down for any period of time [14]. In other situations, although continuous availability may not be safety or business critical, it is simply inconvenient to interrupt the execution of a piece of software in order to perform an upgrade.

Unanticipated software evolution tries to allow for the evolution of systems in response to changes in requirements that were not known at the initial design time. There have been a number of attempts at solving these problems at the levels of evolving methods and classes [5, 7], components [11] and services [15]. In this paper we consider an approach to software evolution at the architectural level, in terms of *plugin* components.

We believe that it is possible to engineer a generalised and flexible plugin architecture which will allow applications to be extended dynamically at runtime. Here we present a model of how components may be assembled in such an architecture based on the interfaces that they present. This model will be used at run-time by a plugin framework to determine the connections that can and should be made between plugins (our implementation of such a framework is detailed in [3]).

The benefits of building software out of a number of modules have long been recognised. Encapsulating certain functionality in modules and exposing an interface evolved into component oriented software development [2]. Components can be combined to create systems. An important difference between plugin based architectures and other component based architectures is that plugins are optional rather than required components. The system should run regardless of whether or not plugin components have been added, but offer varying degrees of functionality depending on what plugins are present. Plugins can be used to address the following issues:

- the need to extend the functionality of a system,
- the decomposition of large systems so that only the software required in a particular situation is loaded,
- the upgrading of long-running applications without restarting,
- incorporating extensions developed by third parties.

Plugins have previously been used to address each of these different situations individually, but the architectures designed have generally been quite specifically targeted and therefore limited. In existing systems, either there are constraints on what can be added, or creating extensions requires a lot of work on the behalf of the developer, for example writing architectural definitions that describe how components can be combined [13]. We believe that it is possible to engineer a more generalised and flexible plugin architecture not requiring the connections between components to be explicitly stated.

Here we describe how formal specification techniques helped us in developing a generalised plugin model that can be used to deal with any of the situations described above. Unlike other plugin models (for example that used by Eclipe [13]), in our model components are matched purely based on information that is available from the code, rather than using meta-data such as an IDL description. In the remainder of the paper we present our model both informally,

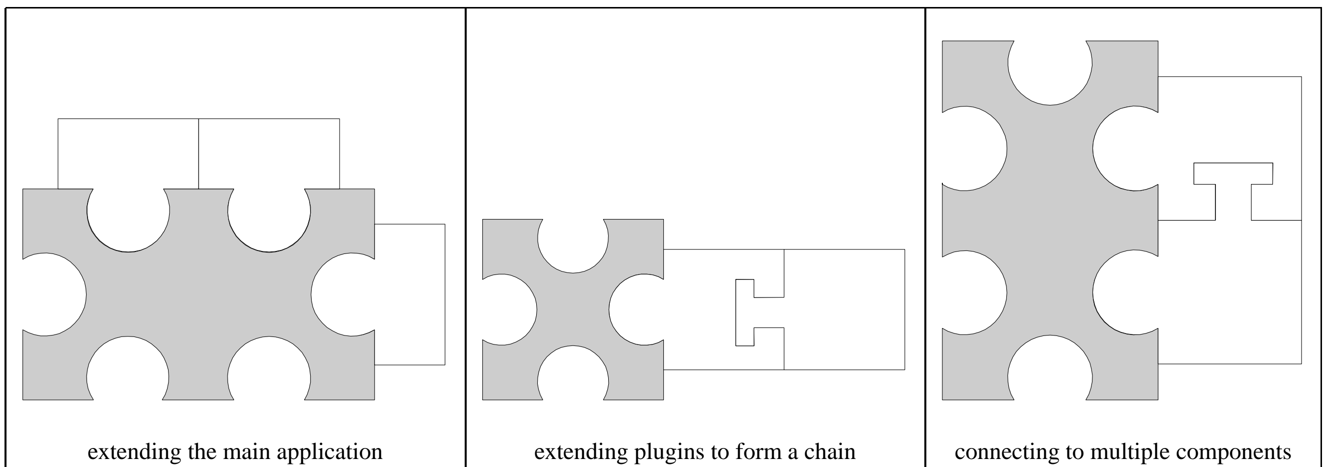



**Figure 1: Some possible configurations of plugins**

based on a familiar analogy, and formally using the specification language Alloy.

## 2. AN INFORMAL MODEL

We think of the way that components fit together in a plugin architecture as being similar to the way that pieces of a jigsaw puzzle fit together. As long as a jigsaw piece has the right shaped peg, it can connect to another piece that has a corresponding hole.

The main application provides a number of holes, into which components providing extra functionality can plug. Plugins are optional components containing collections of classes and interfaces. The holes represent an interface known to the main application, and the pegs represent classes in the plugin components that implement this interface. The interface defines the signatures of methods in the class. If an application has an interface that allows other components to extend it, and a plugin contains a class that implements this interface, a connection can be made between them. The peg will fit into the hole. This situation, adding components to a central application, is shown in the first example Figure in 1.

Thinking about plugins in this way, it becomes clear that some other more sophisticated configurations would be possible if we allow plugin components to have holes as well as pegs, *i.e.* if we allow plugins to extend other plugins rather than only allowing them to extend the main application. We can then have chains of plugins as shown in the middle example in Figure 1. An example of this situation might be if the main application were a word processor, which was extended by plugging in a graphics editor, and this graphics editor was in turn extended by plugging in a new drawing tool.

It is possible that a component has several holes and pegs of different shapes (probably the most common situation in traditional jigsaw puzzles). This can lead to more complicated configurations of components, such as those shown in the rightmost example in Figure 1. Such a configuration might be useful in a situation where the main application was, say, an integrated development environment, the first plugin was a help browser, and the second a debugging tool. The debugging tool plugs into the the main application, but also into the help browser so that it can contribute help relevant to debugging. In this way the help browser can display help provided by all of the different tools in the IDE, with the help being stored locally in each of the separate tools. It is clear that we cannot represent all possible configurations of plugins using these simple planar jigsaw representations, but they provide a useful metaphor for thinking about what might be possible.

If we think once again about the first case, then it seems that we should be able to keep on adding plugins to the application as long as they implement the right interface, but there might be cases where we want to put limits on the number of plugins that can be attached. This might be the case when each plugin that is added consumes a resource held by the main application, of which a limited quantity is available. Cardinality constraints can also be employed to constrain the shapes that the configuration can take.

To see the effect of using cardinalities, consider a main application which accepts a certain type of plugin, without a restriction on how many plugins can be added. If three compatible plugins are added, all three will be loaded and connected to the system. If, however, we change the cardinality of the interface to be $\leq 2$, *i.e.* any number up to a maximum of two, after two plugins have been added, a third cannot be. It might be possible to remove plugin 1 or 2, and to replace it with plugin 3, but it is not possible to plug in all three at the same time. In practice though it seems that the two cardinalities used most often will probably be $\leq 1$ and “any number”.

Revisiting the chaining patterns that we saw earlier (see the second example in Figure 1), but employing cardinalities, we can chain together a number of different components of the same type, by having each provide and accept one peg of the same shape (limiting the number of pegs accepted requires a cardinality constraint - see Figure 2). This is almost like a Decorator pattern [6] for components. A decorator conforms to the interface of the component it decorates so that it adds functionality but its presence is transparent to the component's clients. Such a situation might be useful if, for instance, we wanted to chain together video filters, each of which took a video stream as an input and provided another stream as an output. Each filter could perform a different transformation (for example converting the image to black and white, or inverting it) but the components could be combined in any order, regardless of the number in the chain. Plugins would allow this configuration to be changed dynamically over time.

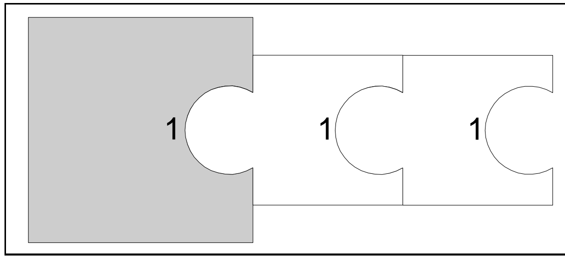



**Figure 2: Chaining with cardinality constraints**

It is our aim to provide the described plugin architectures in self-assembling systems [8]. It should be possible to introduce new components over time. For each additional component the system should make connections to join it to the existing system in accordance with its accepted and provided interfaces. It should not be necessary for the user or developer to provide extra information about how or where the component should be connected, as they may not have total information about the current configuration, or they may just want to delegate responsibility for managing the configuration to the system itself. The plugin framework should be able to assemble the components according to the types of the classes they contain.

Figure 3 shows a possible configuration of a video replay application. The main application displays video streams which are supplied by plugin components. The mixer component mixes two video streams into one, so can be used to add subtitles to a film. In the figure a mixer and a set of subtitles have been added to the application, and a film source is about to be added. The film source could connect either to the mixer or directly to the video player. In the first case, the subtitles will be applied to the film, in the second case the film and the subtitles will be displayed separately. We would like to be able to ensure that the behaviour desired by the provider of the film component is implemented or at very least to predict what will happen in this case. We need to know that the same thing will happen if the same components are combined on different occasions.

It is desirable that the behaviour of self-assembling systems can be made to be deterministic: it should be possible to determine what connections will be made when a certain component is added to a certain configuration. To ensure that this is the case, provision needs to be made for defining a strategy to decide between different possible bindings in a predictable way. The technique we use for this is to allow strategies for deciding between different possible bindings to be provided in the form of preference functions written by plugin developers.

## 3. A FORMAL MODEL

Before implementing a framework to support applications that are extensible with plugins, we developed a formal specification for the system in Alloy [9]. Alloy is a lightweight notation that supports the description of systems that have relational structures. The systems that we wish to describe are concerned with sets of linked components, so Alloy is a particularly appropriate language. The notation allows us to write any first-order logical expression plus transitive closure. In addition to providing language constructs that fit our domain, Alloy has the advantage that specifications are able to be analysed automatically. Analysis is supported by, the Alloy

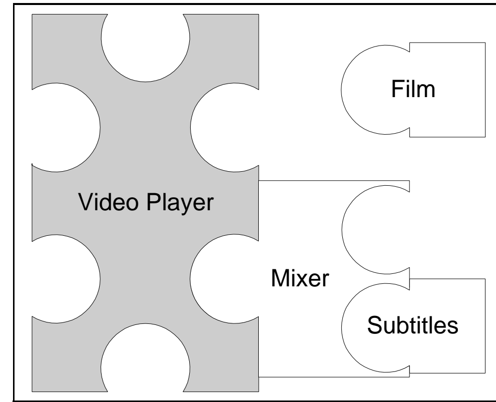



**Figure 3: Non-determinism**

Constrain Analyser (ACA) [4]. This tool allows us to check our Alloy models for consistency and to generate example situations which we may not have considered. Uncovering the possibility of such unexpected behaviour early in the development process allows us to refine the specification to deal with it, rather than having to do much more expensive maintenance, as would be the case if problems were discovered after implementation.

Using Alloy allows us to represent formally the way in which plugin components can fit together, and what happens when a new component is added to the system. In the case that we have written inconsistent constraints, the analyser will report that it could not generate an example that satisfies the constraints that we have specified.

The ACA tool provides a visualiser which will display example structures graphically. This representation is easy to interpret. We can see how the components have been joined together to form a system. The figures in this paper were generated by this visualisation tool (with minor hand editing of labels to make the examples easier to understand). The visualisation tool is quite flexible, allowing us to omit parts of the model and to show labels either within an object or with an arrow from the object. In Figure 4 we have used both techniques, purely for clarity.

In the text of this paper we present the model in first order logic for readability, and again in Alloy in the Appendix.

One of the ideas described in the previous section is that the number of each type of plugin component allowed may be explicitly defined. This is quite a complicated property and so we first model plugins without it and then extend the model to include cardinality constraints.

### 3.1 A basic model

The artifacts we model could be created by a compiler for an object oriented language with name equivalence. So they could be created by a Java or C# compiler.

Classes are defined in terms of the interfaces they implement and whether or not they are `abstract`. The type interface $\mathcal{I}$ is atomic.[1] We use the notation '?' to mean `abstract` may optionally be present

[1] For a declared type $\mathcal{T}$, $t \in \mathcal{T}$ and $t : \mathcal{T}$ will be used interchangeably.

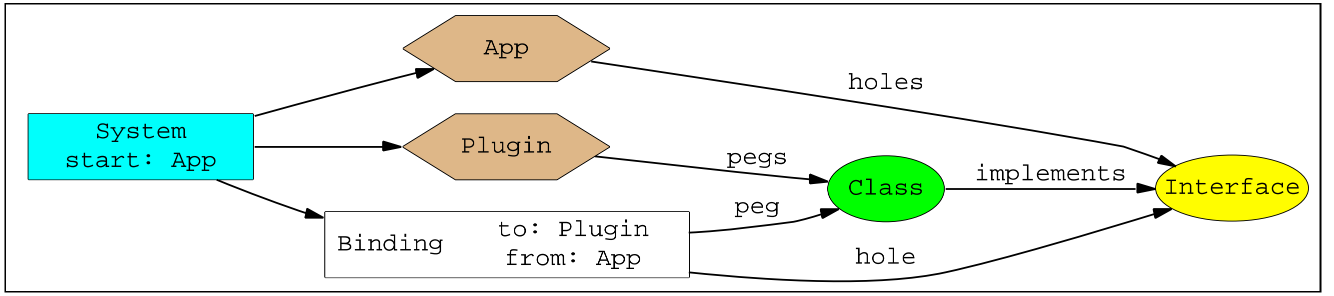



**Figure 4: One component added**

in a given class. As they have already been successfully compiled we know that classes must implement the interfaces they say they implement.

DEFINITION 1. *A class $cl : \mathcal{CL}$ is defined as:*

$$cl = \{\mathsf{implements} : \mathcal{P}(\mathcal{I}), \mathsf{abstract?} : String\}$$

Components $\mathcal{C}$ are just sets of classes and sets of interfaces. The classes constitute what the component provides and the interfaces are what the component can accept.[2]

DEFINITION 2. *A component $c : \mathcal{C}$ is defined as:*

$$c = \{\mathsf{pegs} : \mathcal{P}(\mathcal{CL}), \mathsf{holes} : \mathcal{P}(\mathcal{I})\}$$

Components need to be connected or bound together. Bindings $\mathcal{B}$ connect classes to interfaces. The component containing the interface has to be different from the component containing the class, so that components cannot plug in to themselves. The need for this constraint was not originally apparent. Considering examples produced by the analyser that did not follow this constraint caused us to add it to the model (see Section 4 for more discussion).

DEFINITION 3. *A binding $b : \mathcal{B}$ is defined as:*

$$b = \{\mathsf{peg} : \mathcal{CL}, \mathsf{to} : \mathcal{C}, \mathsf{hole} : \mathcal{I}, \mathsf{from} : \mathcal{C}\}$$

*such that:*

$$(\mathsf{to} \neq \mathsf{from}) \wedge (\mathsf{peg} \in \mathsf{to.pegs}) \wedge (\mathsf{hole} \in \mathsf{from.holes})$$

A System $\mathcal{S}$ consists of a set of components, a set of bindings between interfaces and classes of the components and a special component, designated start. The start component must contain at least one interface or there would be no way of ever extending a system containing it as the first component. All other components must contain some classes in order that they can provide some extra functionality to the system. An interface cannot be bound to a given class more than once (the same class in a different component is taken to be a different class).

[2] In the implementation of this model, components also include sets of resources, but these would add nothing to our model so we have omitted them.

DEFINITION 4. *A system $s : \mathcal{S}$ is defined as:*

$$s = \{\mathsf{comps} : \mathcal{P}(\mathcal{C}), \mathsf{bindings} : \mathcal{P}(\mathcal{B}), \mathsf{start} : \mathcal{C}\}$$

*such that:*

$$\begin{array}{l} \mathsf{start} \in \mathsf{comps} \\ \exists b : \mathcal{B}.(\mathsf{start} = b.\mathsf{from}) \vee (\#\mathsf{comps} = 1) \\ \forall c \in \mathsf{comps}.(c.\mathsf{pegs} \neq \emptyset \vee c = \mathsf{start}) \\ \forall b_1, b_2 : \mathcal{B}.(((b_1.\mathsf{from} = b_2.\mathsf{from}) \wedge (b_1.\mathsf{hole} = b_2.\mathsf{hole}) \wedge \\ \quad (b_1.\mathsf{to} = b_2.\mathsf{to}) \wedge (b_1.\mathsf{peg} = b_2.\mathsf{peg})) \Rightarrow (b_1 = b_2)) \end{array}$$

Figure 4 shows a system with an application and a single plugin. In this system the starting component is App, which has a single interface with one method header. Plugin is added and a binding is formed from App to Plugin because Plugin contains Class, which implements Interface.

Classes and interfaces cannot exist in isolation. Every class and every interface is associated with a component. Similarly, bindings and components are always associated with systems and all components (with the possible exception of when a system contains exactly one component) are bound to other components. These constraints were not thought about explicitly before we started modelling our proposed systems. Each property has to be built into any framework that implements our model so that we create systems that behave in the way predicted by our model.

PROPERTY 1 (NO ORPHANS IN ANY $s : \mathcal{S}$).

$$\begin{array}{l} \forall i : \mathcal{I}.\exists c : \mathcal{C}.(i \in c.\mathsf{holes}) \\ \forall cl : \mathcal{CL}.\exists c : \mathcal{C}.(cl \in c.\mathsf{pegs}) \\ \forall b : \mathcal{B}.\exists s : \mathcal{S}.(b \in s.\mathsf{bindings}) \\ \forall c : s.\mathsf{comps}.(\exists b : \mathcal{B}.c = b.\mathsf{to} \vee c = b.\mathsf{from}) \\ \quad \vee(\#s.\mathsf{comps} = 1) \end{array}$$

The addition of a plugin component to an existing system needs to be modelled. A component can only be added if it has a class that is not abstract that implements an interface in the existing system. But before we look at a function to add a new component to a system, we first will need to test whether two components with an associated interface and class can be bound at all.

DEFINITION 5 (canBind).

$$\begin{array}{ll} \mathsf{canBind} & \subseteq \mathcal{CL} \times \mathcal{C} \times \mathcal{I} \times \mathcal{C} \\ \mathsf{canBind} & (cl, c', i, c) \iff (i \in c.\mathsf{holes}) \wedge (cl \in c'.\mathsf{pegs}) \wedge \\ & (c' \neq c) \wedge (\mathsf{abstract} \notin cl) \wedge i \in cl.\mathsf{implements} \end{array}$$

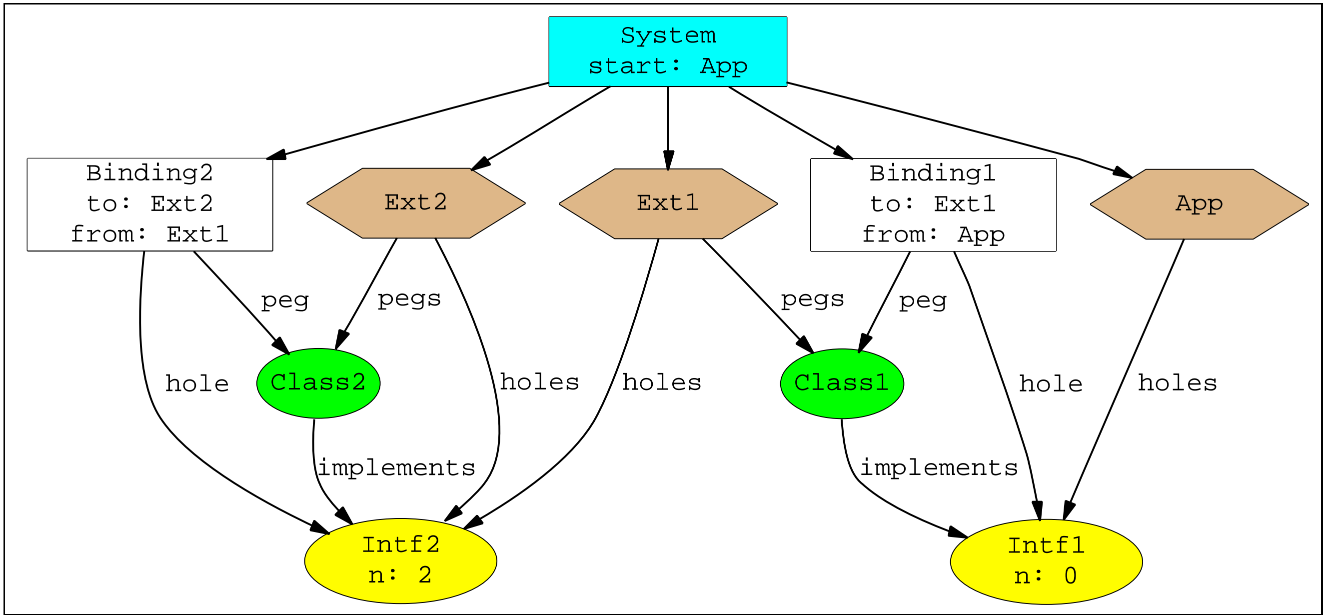



**Figure 5: Several components forming a system**

If a component can be bound to another component in a system, then it can be added to that system. Otherwise trying to add such a component will have no effect on the system.

DEFINITION 6 (ADDITION FUNCTIONS).

*A function* add : $(\mathcal{S}, \mathcal{C}) \longrightarrow \mathcal{S}$ *is an addition function iff*

$$\begin{aligned}
&\mathsf{add}(s, c) = s' \Rightarrow \\
&(\exists c' \in s.\mathsf{comps}.\exists i : \mathcal{I}.\exists cl : \mathcal{CL}.\mathsf{canBind}(cl, c, i, c') \\
&\quad \Rightarrow s' = \{s.\mathsf{comps} \cup \{c\}, s.\mathsf{bindings} \cup \{cl, c, i, c'\}, s.\mathsf{start}\}) \\
&\qquad \vee \\
&(\forall c' \in s.\mathsf{comps}.\neg\mathsf{canBind}(_, c', _, c) \Rightarrow s' = s)
\end{aligned}$$

If there is more than one candidate for $c'$ then there could be a set of possible addition functions each capable of performing the appropriate operation. In order for the system to be deterministic, so that behaviour holds no surprises, we need to choose a single addition function and use it. In section 3.3 we show how to do this.

## 3.2 Extending the Model with Cardinality Constraints

In the model described so far, the number of plugins that can be bound to a particular interface simultaneously is not prescribed. A given interface may have any number of classes bound to it. This is not always what is required. Sometimes the number of classes that can be bound to an interface is fixed. Perhaps for a specific interface only one class should be bound to it. At the other extreme an interface may let any number of classes be bound to it. For this model the numbers will be defined to be the natural numbers (including 0) extended with an infinite number.[3]

[3]In a finite Alloy model a natural number larger than the scope for which the model is analysed will have the same effect on binding as an infinite number would.

DEFINITION 7. *The numbers* $\mathcal{N}$ *are defined as:*

$$\mathcal{N} = \mathcal{N}at \cup \{\infty\}$$

Interfaces need to be extended with the number of classes that can be bound to them.

DEFINITION 8. *A numinterface* $ni : \mathcal{NI}$ *is defined as:*

$$ni = \{i : \mathcal{I}, n : \mathcal{N}\}$$

Numinterfaces need to replace interfaces throughout the definitions and in the NO ORPHANS property. More importantly, the definition of add needs to be changed to take the numbering into account. Firstly, a class can only be bound to an interface if the number associated with that interface is not zero. Secondly, when a new component is added, the number associated with the relevant interface should be decremented.

DEFINITION 9 (dec).

$$\mathsf{dec} : (\mathcal{C}, \mathcal{NI}) \longrightarrow \mathcal{C}$$

$$\mathsf{dec}(c, (i, n)) = \begin{cases} \{\ c.\mathsf{pegs}, \\ \quad \{c.\mathsf{holes} \setminus (i, n) \\ \quad \cup (i, n-1)\} \\ \} & \textbf{if } n \neq 0 \\ c & \textbf{if } n = 0 \end{cases}$$

DEFINITION 10 (ADDITION FUNCTIONS).

*A function* $\mathsf{add} : (\mathcal{S}, \mathcal{C}) \longrightarrow \mathcal{S}$ *is an addition function iff*

$$
\begin{array}{l}
\mathsf{add}(s, c) = s' \Rightarrow \\
(\exists c' \in s.\mathsf{comps}.\exists (i, n) : \mathcal{NI}.\exists cl : \mathcal{CL}.\mathsf{canBind}(cl, c, i, c') \\
\quad \Rightarrow s' = \{ \quad s.\mathsf{comps} \setminus \{c'\} \cup \{\mathsf{dec}(c', (i, n))\} \cup \{c\}, \\
\qquad\qquad s.\mathsf{bindings} \cup \{cl, c, i, c'\}, \\
\qquad\qquad s.\mathsf{start}\}) \\
\qquad\qquad \vee \\
(\forall c' \in s.\mathsf{comps}.\neg \mathsf{canBind}(_, c', _, c) \Rightarrow s' = s)
\end{array}
$$

Figure 5 was produced by the Alloy model in the Appendix. This is the Alloy version of our model including numbers. The figure shows an application extended by a chain of components, as in the second example in Figure 1. Where n : 0 appears in the diagram it means that no more classes can be bound to this interface and n : 2 means that two more classes can be bound to this interface.

### 3.3 Removing the Nondeterminism

The final step in producing a model that is suitable for implementation is to remove the nondeterminism caused by not having a unique addition function. We need somehow to only bind to the *best* component if there is a choice of several components to which the new plugin could be bound.

Only the designer of the plugin will know, given two components that it is possible to plug in to, which would be the best choice. We need a function prefer, which the plugin developer can implement saying for every suitable pair of components, which of the two components should be bound to. If the developer does not care (it does not matter which component a plugin is connected to) then they do not need to specify a prefer function, and the binding will happen non-deterministically as in the previous case.

DEFINITION 11 (prefer).

$$
\begin{array}{l}
\mathsf{prefer} : (\mathcal{C}, \mathcal{C}, \mathcal{C}) \longrightarrow \mathcal{C} \\
\mathsf{prefer}(addc, this, that) = \begin{cases} this & \text{developer's choice} \\ that & \text{developer's choice} \end{cases}
\end{array}
$$

*such that for each* $addc \in \mathcal{C}$ prefer *induces a total order on the binding candidates.*

We next find the set of components that a given component could possibly be bound to.

DEFINITION 12 (match).

$$
\begin{array}{l}
\mathsf{match} : (\mathcal{S}, \mathcal{C}) \longrightarrow \mathcal{P}(\mathcal{C}) \\
\mathsf{match}(s, c) = \{c' \mid c' \in s.\mathsf{comps}.\exists cl : \mathcal{CL}.\exists (i, n) : \mathcal{NI}. \\
\qquad\qquad\qquad \mathsf{canBind}(cl, c, i, c')\}
\end{array}
$$

Given prefer we can find the best component, amongst all those that are possible (match), to bind the new plugin to.

DEFINITION 13 (best).

$$
\begin{array}{l}
\mathsf{best} : (\mathcal{S}, \mathcal{C}) \longrightarrow \mathcal{C} \\
\mathsf{best}(s, c) = c' \Longrightarrow \quad \forall c'' \in \mathsf{match}(s, c). \\
\qquad\qquad\qquad\qquad (c'' \neq c') \Longrightarrow (c' = \mathsf{prefer}(c, c', c''))
\end{array}
$$

Given best, we can now rewrite add so that it is deterministic. If there is no component to bind to the new component then the system without the plugin is returned. If there is one or more components that can be bound then the best one is chosen.

DEFINITION 14 (add).

$$
\begin{array}{l}
\mathsf{add} : (\mathcal{S}, \mathcal{C}) \longrightarrow \mathcal{S} \\
\mathsf{add}(s, c) = \begin{cases} \{ \quad s.\mathsf{comps} \setminus \{c'\} \cup \{\mathsf{dec}(c', (i, n))\} \cup \{c\}, \\ \quad\;\; s.\mathsf{bindings} \cup \{cl, c, i, c'\}, \\ \quad\;\; s.\mathsf{start} \\ \} & \textbf{if } pre \\ s & \textbf{otherwise} \end{cases} \\
\textbf{where } pre = \quad \exists c' \in s.\mathsf{comps}.\exists (i, n) : \mathcal{NI}.\exists cl : \mathcal{CL}. \\
\qquad\qquad ((c' \neq c) \wedge (n \neq 0) \wedge \\
\qquad\qquad\quad \mathsf{canBind}(cl, c, i, c') \wedge (c' = \mathsf{best}(s, c))
\end{array}
$$

## 4. DISCUSSION

By writing an Alloy specification incrementally, and using the ACA tool to generate examples of the system's behaviour at each stage, several situations were uncovered where we had not constrained the specification strictly enough, resulting in undesirable behaviour.

Initially we had not explicitly stated that plugins cannot fill their own holes. The analyser produced an example where one component had a hole and also a matching peg which was bound to the hole. This sparked a discussion as to whether such behaviour was desirable or not. As the intention of holes is that they provide extension points where other components can be bound, we added a constraint to the model that the two components connected by a binding must not be the same component. In this way, working with a formal specification and an analysis tool led us to discuss issues that we had not considered when working with our informal model.

Another situation that came up early on, was one in which several separate groups of components were produced. Each group was connected internally, but not connected to the other groups. As execution starts in the first component, only those components that are transitively connected to the starting component will extend the base application. We therefore amended the NO ORPHANS property, so that there can be no components in the system that do not have a transitive link back to the start component.

In the model we have presented here, we have assumed that the language in which plugins are implemented will be in the style of Java or C# where the interfaces implemented by a class are explicitly named, and matched by name. Therefore in the model a class can just contain a set of interfaces which it implements, rather than us modelling all of the methods in the class and the interface. We assume that the code in plugins has passed through a compiler and so any class that says it implements an interface does in fact define the necessary methods.

If we wanted to model the implementation of plugins in a language with structural typing, where implemented interfaces are not explicitly named, but classes and interfaces are matched based on the methods that they contain, we could simply change the definitions of classes and interfaces, and write a property `implements` to check one against the other. Otherwise the behaviour of the model and the system should be unaffected.

## 5. RELATED WORK

There are several systems currently in existence that use plugin components as an extension mechanism. Java Applets [1] allow code to be downloaded dynamically and run in a Java-enabled web browser. The system is not particularly flexible, as all applets have to have be derived from a particular superclass, and the system cannot be used for extending applications in general.

The Eclipse platform for IDEs [13] uses plugins to allow for the addition of extra functionality. However, plugins are only detected on start-up and cannot be added to the system while it is running.

The work described by Mayer on Lightweight Application Development [12] involves a technique for using plugins with a variety of applications, but only deals with connecting extensions directly to the main application, rather than the more complex configurations that we consider.

The PluggableComponent [16] architecture features a registry to manage the different types of PluggableComponent. The registry is used by a configuration tool to provide a list of available components that administrators can use to configure their applications, so configuration is human driven, where our approach aims at automatic configuration without total knowledge of the system. As with Applets, all PluggableComponents are derived from the PluggableComponent base class, limiting flexibility of what can be used as a plugin.

There have been various attempts at formalising component based systems, for instance Jackson and Sullivan's modelling of COM in Alloy [10]. The PACC group at the SEI have been working on Prediction Enabled Component Technologies (PECT [17]). Their work aims to enable the prediction of properties of compositions of components such as latency, and to constrain the assembly of systems to configurations where certain properties hold.

## 6. CONCLUSIONS

We have presented a model for a system of plugin components. Developing and formalising the model caused us to consider several issues relating to what sorts of behaviours and configurations of plugins should and should not be allowed. Using the Alloy analyser helped us by allowing us to visualise different configurations that could occur with our current model. This helped us to make design decisions and refine the model further.

We have implemented a framework in Java that uses the model described here, and used it to build several applications that can be configured and extended using plugin technology. Details of the implementation can be found in [3].

In [14] Oreizy *et al* identify three types of architectural change that are desirable at runtime: component addition, component removal and component replacement. In the future we hope to extend the model presented here to cover all of these cases and to implement such a system.

## 7. ACKNOWLEDGMENTS

We gratefully acknowledge the support of the European Union under grant STATUS (IST-2001-32298). We would also like to acknowledge the SLURP group at Imperial College London, especially Sophia Drossopoulou and Matthew Smith for their help with the formal model, and Matthew again for his help in producing the diagrams that appear in this paper.

## 8. REFERENCES

[1] Applets. Technical report, Sun Microsystems, Inc., java.sun.com/applets/, 1995-2003.

[2] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Pub Co, 1997.

[3] R. Chatley, S. Eisenbach, and J. Magee. Painless Plugins. Technical report, Imperial College London, www.doc.ic.ac.uk/~ rbc/writings/pp.pdf, 2003.

[4] D. Jackson, I. Schechter, and I. Shlyakhter. *Alcoa: the Alloy Constraint Analyzer*, pages 730–733. ACM Press, Limerick, Ireland, May 2000.

[5] M. Dmitriev. HotSwap Client Tool. Technical report, Sun Microsystems, Inc., www.experimentalstuff.com/Technologies/HotSwapTool/index.html, 2002-2003.

[6] E. Gamma, R. Helm, R. Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Pub Co, 1995.

[7] P. S. G. Bierman, M. Hicks and G. Stoyle. Formalising dynamic software updating. In *Second International Workshop on Unanticipated Software Evolution at ETAPS '03*, 2003.

[8] D. Garlan, J. Kramer, and A. Wolf, editors. *Proc. of the First ACM SIFGOSFT Workshop on Self-Healing Systems*. ACM Press, November 2002.

[9] D. Jackson. Micromodels of Software: Lightweight Modelling and Analysis with Alloy. Technical report, M.I.T., sdg.lcs.mit.edu/ dng/, February 2002.

[10] D. Jackson and K. Sullivan. COM Revisited: Tool Assisted Modelling and Analysis of Software Structures. In *In proc. ACM SIGSOFT Conf. Foundations of Software Engineering*, 2000.

[11] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE TSE*, 16(11):1293–1306, November 1990.

[12] J. Mayer, I. Melzer, and F. Schweiggert. Lightweight plug-in-based application development, 2002.

[13] Object Technology International, Inc. Eclipse Platform Technical Overview. Technical report, IBM, www.eclipse.org/whitepapers/eclipse-overview.pdf, July 2001.

[14] P. Oriezy, N. Medvidovic, and R. Taylor. Architecture-based runtime software evolution. In *ICSE '98*, 1998.

[15] M. Oriol. Luckyj: an asynchronous evolution platform for component-based applications. In *Second International Workshop on Unanticipated Software Evolution at ETAPS '03*, 2003.

[16] M. Völter. Pluggable Component - A Pattern for Interactive System Configuration. In *EuroPLoP '99*, 1999.

[17] K. C. Wallnau. A technology for predictable assembly from certifiable components (pacc). Technical report, Software Engineering Insitute, http://www.sei.cmu.edu/pacc/publications.html, 2003.

## Appendix: The Model in Alloy

```
module Plugins

open std/ord

sig String {}
sig Number {}
sig Interface{}

sig Class {
   implements : set Interface,
   abstract : option String
}

sig NumInterface extends Interface{
   n : Number
}

sig Component {
   pegs  : set Class,
   holes : set NumInterface
}

sig Binding {
   hole : NumInterface,
   from : Component,
   peg  : Class,
   to   : Component
}{
    to != from
    hole in from.holes
    peg in to.pegs
}

sig System {
   components : set Component,
   bindings   : set Binding,
   start  : Component
}{
   one start
   start in components
   some start.holes
   some bindings => some b in bindings { start = b.from }
   all c in components { c != start => some c.pegs }
}


fact noOrphans {
   all i : Interface | some c : Component { i in c.holes }
   all cl : Class | some c : Component { cl in c.pegs }
   all b : Binding | some s : System { b in s.bindings }
   all c : Component | #Component = 1 || some b : Binding { c = b.to || c = b.from }
   all c : Component | some s : System {c in s.components}
}

fun dec( c, c' :Component, i : NumInterface ){
   i in c.holes && i.n != Ord[Number].first
   some i' : NumInterface {
   i'.n = OrdPrev(i.n)
   c'.pegs = c.pegs && c'.holes = c.holes - i + i'
   }
}
```

```
fun add( s, s' : System, c : Component ) {
  some c' in s.components | some  i in c'.holes | some cl in c.pegs {
    ( c != c' ) && no cl.abstract && i in cl.implements
    s'.start = s.start
    some c'' : Component {
        dec(c',c'',i)
        s'.components = s.components + c  + c''
    }
    one b : Binding {b.hole = i && b.from = c' && b.peg = cl && b.to = c &&
        s'.bindings = s.bindings + b
    }
   }
}
```

# Form-based Software Composition

Markus Lumpe
Iowa State University
Department of Computer Science
113 Atanasoff Hall
Ames, USA
lumpe@cs.iastate.edu

Jean-Guy Schneider
School of Infomation Technology
Swinburne University of Technology
P.O. Box 218
Hawthorn, VIC 3122, AUSTRALIA
jschneider@swin.edu.au

## ABSTRACT

The development of flexible and reusable abstractions for software composition has suffered from the inherent problem that reusability and extensibility are limited due to position-dependent parameters. To tackle this problem, we have defined *forms*, immutable extensible records, that allow for the specification of compositional abstractions in a language-neutral and robust way. In this paper, we present a theory of forms and show how forms can be used to represent components based on software artifacts developed for the .NET framework.

## 1. INTRODUCTION

In recent years, component-oriented software technology has become the major approach to facilitate the development of evolving systems [9, 17, 24, 28]. The objective of this technology is to take elements from a collection of reusable software components (i.e., *components-off-the-shelf*) and build applications by simply plugging them together.

Currently, component-based programming is mainly carried out using mainstream object-oriented languages. These languages seem to offer already some reasonable support for component-based programming (e.g. encapsulation of state and behavior, inheritance, and late binding). Unfortunately, object-oriented techniques are not powerful enough to provide flexible and typesafe component composition and evolution mechanisms, respectively. We can identify especially (a) a lack of abstractions for building and adapting class-like components in a framework or domain specific way, (b) a lack of abstractions for defining cooperation patterns, and (c) a lack of support for checking the correctness of compositions.

In order to address the key problems, various researchers have argued that it is necessary to define a language specially designed to compose software components and to base this language on a well-defined formal semantic foundation [2, 9, 12, 13, 18, 19]. But what kind of formalism should be used as a semantic foundation?

There are several plausible candidates that can serve as computational models for component-based software development. The $\lambda$-calculus, for example, has the advantage of having a well-developed theoretical foundation and being well-suited for modeling encapsulation, composition and type issues [5], but has the disadvantage of saying nothing about concurrency or communication. Process calculi such as CCS [15] and the $\pi$-calculus [16] have been developed to address just these shortcomings. Early work in the modeling of concurrent objects [20, 21] has proven CCS to be an expressive modeling tool, except that dynamic creation and communication of new communication channels cannot be directly expressed and that abstractions over the process space cannot be expressed within CCS itself, but only at a higher level. These shortcomings have been addressed by the $\pi$-calculus, which allows new names to be introduced and communicated much in the same way the $\lambda$-calculus introduces new bound names.

Unfortunately, even though both the $\lambda$-calculus and the $\pi$-calculus can be used to model composition mechanisms [27], they are inconvenient for modeling general purpose compositional abstractions due to the dependence on positional parameters. In fact, the need to use position dependent parameters results in a limited reusability and extensibility of the defined abstractions.

Dami has tackled a similar problem in the context of the $\lambda$-calculus, and has proposed $\lambda$N [6, 7], a calculus in which parameters are identified by names rather than by positions. The resulting flexibility and extensibility can also be seen, for example, in XML/HTML forms, whose fields are encoded as named (rather than positional) parameters, in Python [30], where functions can be defined to take arguments by keywords, in Visual Basic [14], where *named arguments* can be used to break the order of possibly optional parameters, and in Perl [32] where it is a common technique to pass a map of name/value pairs as argument to a function or method.

*Forms* are immutable extensible records that define finite mappings from keys to values. They address the inherent problem that reusability and extensibility of abstractions are limited due to position-dependent parameters [11]. We argue that forms provide the means for a unifying concept

to define robust and extensible compositional abstractions. Unlike classical records, however, forms are abstract, that is, forms are used to define mappings from keys to abstract values. This is a generalization of an earlier work on forms [11, 26] and will allow us to study forms as an environment-independent framework. In order to actually use these compositional abstractions, they have to be instantiated in a concrete target system like the $\pi\mathcal{L}$-calculus [11] or the .NET framework [22]. In this paper, we will outline the basic ideas and formalisms for first-order forms, sketch some of the important issues in relation to form equivalence and normalization, and illustrate how forms can be used as a semantic foundation for component composition.

The remainder of this paper is organized as follows: in section 2, we present first-order forms. In section 3, we develop a semantics of forms. In section 4, we illustrate, how forms can be used to represent component interfaces and component interface composition of components written in the .NET-aware language C#. We conclude with a summary of the main observations and a discussion about future work in section 5.

## 2. FORMS

Forms are finite variable-free mappings from an infinite set of labels denoted by $\mathcal{L}$ to an infinite set of abstract values denoted by $\mathcal{V}$. The set of abstract values is a set of distinct values. We do not require any particular property except that equality and inequality are defined for all abstract values. In fact, programming values like *Strings*, *Integers*, *Names*, and even *Objects* and *Classes* are elements of $\mathcal{V}$.

The set $\mathcal{V}$ contains a distinguished element $\mathcal{E}$ – the empty value. In the context of software composition, the empty value denotes the lack of a *component service*. That is, if a given label, say $l$, is bound to the empty value, then the corresponding component service is either currently unavailable or not defined at all.

We use $F, G, H$ to range over the set $\mathcal{F}$ of forms, $l, m, n$ to range over the set $\mathcal{L}$ of labels, and $a, b, c$ to range over the set $\mathcal{V}$ of abstract values. The set $\mathcal{F}$ of forms is defined as follows:

| | | | |
|---|---|---|---|
| $F$ | $::=$ | $\langle\rangle$ | *empty form* |
| | $\mid$ | $F\langle l = V\rangle$ | *abstract binding extension* |
| | $\mid$ | $F \cdot F$ | *polymorphic extension* |
| | $\mid$ | $F\backslash F$ | *polymorphic restriction* |
| | $\mid$ | $F \rightarrow l$ | *form dereference* |
| $V$ | $::=$ | $S$ | *abstract scalar value* |
| | $\mid$ | $F$ | *nested form* |
| $S$ | $::=$ | $\mathcal{E}$ | *empty value* |
| | $\mid$ | $a$ | *abstract value* |
| | $\mid$ | $F_l$ | *abstract projection* |

Every form is derived from the *empty form* $\langle\rangle$, which denotes an empty component interface (i.e., a component that does not define any service). The *abstract binding extension* $F\langle l = V\rangle$ extends a given form $F$ with exactly one binding $\langle l = V\rangle$ that either adds a fresh service, named $l$, or redefines an existing one. Using *polymorphic extension*, we can add or redefine a set of services. Polymorphic extension is similar to asymmetric record concatenation [4]. In fact, if the forms $F$ and $G$ both define a binding for the label $l$, then only $G's$ binding will be accessible in the polymorphic extension $F \cdot G$. The *polymorphic restriction* $F\backslash G$ denotes a form that is restricted to all bindings of $F$ that do not occur in $G$ with the exception that all bindings of the form $\langle l = \mathcal{E}\rangle$ and $\langle l = \langle\rangle\rangle$ in $G$ are ignored. Finally, the *form dereference* $F \rightarrow l$ denotes the form that is bound by label $l$ in $F$. Form dereference can be used to extract a form that occurs nested within an enclosing form.

In form expressions, an abstract binding extension has precedence over a polymorphic extension, a polymorphic extension has precedence over a polymorphic restriction, which in turn has precedence over form dereference. A sequence of two or more polymorphic extensions is left associative, that is, $F_1 \cdot F_2 \cdot F_3$ is equivalent to $(F_1 \cdot F_2) \cdot F_3$. The same applies to polymorphic restriction. Parenthesis may be used in form expressions in order to enhance readability or to overcome the default precedence rules.

Forms denote *component interfaces* and *component interface composition*. A component offers services using *provided ports*, and may require services using *required ports*. If a component offers a service, then this service is bound by a particular label in the form that represents the component interface. For example, if a component can provide a service $A$ and we want to publish this service using the port name *ServiceA*, then the corresponding component interface contains a binding $\langle ServiceA = A\rangle$. In the component interface, the service $A$ is abstract. Therefore, the client and the component (service provider) have to agree on an interpretation of service $A$ prior to interaction. From an external observer's point of view, the interaction between the client and the component involves an opaque entity.

Required services are denoted by *abstract projections*. Given a form $F$, used as a deployment environment [1], and a component, which requires a service named $l$, we use $F_l$ to denote the required service bound by $l$ in the deployment environment $F$.

Using *binding extension* and *projection*, it is possible to construct "flat" data structures that denote both provided and requires services. However, the composition of two or more components often results in a name clash of their port names. Moreover, it is sometimes desirable to maintain the original structure of the components in the composite. To solve these kinds of problems, it is necessary to define auxiliary abstractions that encapsulate components as services. For example, a component *ComponentA* can be represented by an auxiliary service *ServiceComponentA*. Using a binding extension $\langle ServiceA = ServiceComponentA\rangle$, we can publish this service. However, this approach requires that clients have to define an additional abstraction to extract *ComponentA* encapsulated by *ServiceComponentA*.

To facilitate the specification of structured component in-

terfaces, forms can also contain *nested forms*. Like abstract values, nested forms are bound by labels. However, a projection of a nested form yields $\mathcal{E}$. The reason for this is that forms represent sets of key-value bindings and not abstract values. To extract a nested form bound by a label $l$ in a form $F$, we use $F \rightarrow\; l$. Note that if the binding involving label $l$ does not denote a nested form, then the actual value of $F \rightarrow\; l$ is $\langle\rangle$ – the empty form.

## 3. SEMANTICS OF FORMS

The underlying semantic model of forms is that of a record data structure. Forms are generic extensible records, where field selection is performed from right-to-left.

The interpretation of forms is defined by an evaluation function $[\![]\!]^F : \mathcal{F} \rightarrow \hat{\mathcal{F}}$, which is a total function from forms to form values. Like forms, form values are finite mappings from an infinite set of labels denoted by $\mathcal{L}$ to an infinite set of abstract values denoted by $\mathcal{V}$. However, form values do not contain any *projections* or *form dereferences*.

We use $\hat{F}, \hat{G}, \hat{H}$ to range over the set $\hat{\mathcal{F}}$ of form values, $l, m, n$ to range over the set $\mathcal{L}$ of labels, and $a, b, c$ to range over the set $\mathcal{V}$ of abstract values. The set $\hat{\mathcal{F}}$ of form values is a subset of $\mathcal{F}$, i.e., $\hat{\mathcal{F}} \subset \mathcal{F}$, and is defined as follows:

$$
\begin{array}{lcll}
\hat{F} & ::= & \langle\rangle & \textit{empty form value} \\
 & | & \hat{F}\langle l = \hat{V}\rangle & \textit{binding extension value} \\
 & | & \hat{F} \cdot \hat{F} & \textit{polymorphic extension value} \\
 & | & \hat{F} \backslash \hat{F} & \textit{polymorphic restriction value} \\
 & & & \\
\hat{V} & ::= & \hat{S} & \textit{abstract scalar value} \\
 & | & \hat{F} & \textit{nested form value} \\
 & & & \\
\hat{S} & ::= & \mathcal{E} & \textit{empty value} \\
 & | & a & \textit{abstract value}
\end{array}
$$

In order to define $[\![]\!]^F$, we need to define two mutually dependent functions to evaluate *projections* and *form dereferences*. The function $[\![]\!] : \hat{\mathcal{F}} \times \mathcal{L} \rightarrow \mathcal{V}$, called *projection evaluation*, is a total function from pairs $(\hat{F}, l) \in \hat{\mathcal{F}} \times \mathcal{L}$ to abstract values $a \in \mathcal{V}$, whereas the function $\langle\!\langle\rangle\!\rangle : \hat{\mathcal{F}} \times \mathcal{L} \rightarrow \hat{\mathcal{F}}$, called *form dereference evaluation*, is a total function from pairs $(\hat{F}, l) \in \hat{\mathcal{F}} \times \mathcal{L}$ to form values $\hat{G} \in \hat{\mathcal{F}}$. We define *projection evaluation* first.

Definition 1. *Let $\hat{F} \in \hat{\mathcal{F}}$ be a form value and $l$ be a label. Then the application of the function $[\![]\!] : \hat{\mathcal{F}} \times \mathcal{L} \rightarrow \mathcal{V}$ to the projection $\hat{F}_l$, written $[\![\hat{F}_l]\!]$, yields an abstract value $a \in \mathcal{V}$ and is inductively defined as follows:*

$$
\begin{array}{rcll}
[\![\langle\rangle_l]\!] & = & \mathcal{E} & \\
[\![(\hat{F}\langle m = \hat{V}\rangle)_l]\!] & = & [\![\hat{F}_l]\!] & \textit{if } m \neq l \\
[\![(\hat{F}\langle l = \hat{V}\rangle)_l]\!] & = & \begin{cases} \hat{V} & \textit{if } \hat{V} \in \mathcal{V} \\ \mathcal{E} & \textit{otherwise} \end{cases} & \\
[\![(\hat{F} \cdot \hat{G})_l]\!] & = & \begin{cases} [\![\hat{G}_l]\!] & \textit{if } [\![\hat{G}_l]\!] \neq \mathcal{E} \vee \langle\!\langle \hat{G} \rightarrow\; l \rangle\!\rangle \neq \langle\rangle \\ [\![\hat{F}_l]\!] & \textit{otherwise} \end{cases} & \\
[\![(\hat{F} \backslash \hat{G})_l]\!] & = & \begin{cases} \mathcal{E} & \textit{if } [\![\hat{G}_l]\!] \neq \mathcal{E} \vee \langle\!\langle \hat{G} \rightarrow\; l \rangle\!\rangle \neq \langle\rangle \\ [\![\hat{F}_l]\!] & \textit{otherwise} \end{cases} &
\end{array}
$$

To illustrate the effect of *projection evaluation*, consider the following examples:

$$
\begin{array}{l}
[\![(\langle\rangle\langle l = a\rangle\langle m = b\rangle)_m]\!] = b \\
[\![((\langle\rangle\langle l = a\rangle\langle m = b\rangle) \backslash (\langle\rangle\langle m = c\rangle))_m]\!] = \mathcal{E} \\
[\![(\langle\rangle\langle l = a\rangle\langle m = \langle\rangle\langle n = c\rangle\rangle)_m]\!] = \mathcal{E} \\
[\![((\langle\rangle\langle l = a\rangle\langle m = \langle\rangle\langle n = c\rangle\rangle) \cdot (\langle\rangle\langle l = b\rangle\langle m = d\rangle))_m]\!] = d
\end{array}
$$

In the form $\langle\rangle\langle l = a\rangle\langle m = b\rangle$ the abstract value $b$ is bound by label $m$. Therefore, $[\![(\langle\rangle\langle l = a\rangle\langle m = b\rangle)_m]\!]$ yields $b$. The second projection evaluation yields $\mathcal{E}$, because $\langle\rangle\langle m = c\rangle$ restricts $\langle\rangle\langle l = a\rangle\langle m = b\rangle$ by hiding the binding involving label $m$. In the third example, the label $m$ binds a nested form. However, nested forms are not abstract values. Therefore, the result of the projection evaluation is $\mathcal{E}$. Finally, since the form $\langle\rangle\langle l = b\rangle\langle m = d\rangle$ is used as a polymorphic extension, the project evaluation of the whole form yields $d$, which is the right-most value bound by label $m$.

Definition 2. *Let $\hat{F} \in \hat{\mathcal{F}}$ be a form value and $l$ be a label. Then the application of the function $\langle\!\langle\rangle\!\rangle : \hat{\mathcal{F}} \times \mathcal{L} \rightarrow \hat{\mathcal{F}}$ to the form dereference $\hat{F} \rightarrow\; l$, written $\langle\!\langle \hat{F} \rightarrow\; l \rangle\!\rangle$, yields a form value $\hat{G} \in \hat{\mathcal{F}}$ and is inductively defined as follows:*

$$
\begin{array}{rcll}
\langle\!\langle \langle\rangle \rightarrow\; l \rangle\!\rangle & = & \langle\rangle & \\
\langle\!\langle (\hat{F}\langle m = \hat{V}\rangle) \rightarrow\; l \rangle\!\rangle & = & \langle\!\langle \hat{F} \rightarrow\; l \rangle\!\rangle & \textit{if } m \neq l \\
\langle\!\langle (\hat{F}\langle l = \hat{V}\rangle) \rightarrow\; l \rangle\!\rangle & = & \begin{cases} \langle\rangle & \textit{if } \hat{V} \in \mathcal{V} \\ \hat{V} & \textit{otherwise} \end{cases} & \\
\langle\!\langle (\hat{F} \cdot \hat{G}) \rightarrow\; l \rangle\!\rangle & = & \begin{cases} \langle\!\langle \hat{G} \rightarrow\; l \rangle\!\rangle & \textit{if } [\![\hat{G}_l]\!] \neq \mathcal{E} \vee \langle\!\langle \hat{G} \rightarrow\; l \rangle\!\rangle \neq \langle\rangle \\ \langle\!\langle \hat{F} \rightarrow\; l \rangle\!\rangle & \textit{otherwise} \end{cases} & \\
\langle\!\langle (\hat{F} \backslash \hat{G}) \rightarrow\; l \rangle\!\rangle & = & \begin{cases} \langle\rangle & \textit{if } [\![\hat{G}_l]\!] \neq \mathcal{E} \vee \langle\!\langle \hat{G} \rightarrow\; l \rangle\!\rangle \neq \langle\rangle \\ \langle\!\langle \hat{F} \rightarrow\; l \rangle\!\rangle & \textit{otherwise} \end{cases} &
\end{array}
$$

To illustrate the effect of *dereference evaluation*, consider the following examples:

$$
\begin{array}{l}
\langle\!\langle (\langle\rangle\langle l = a\rangle\langle m = b\rangle) \rightarrow\; m \rangle\!\rangle = \langle\rangle \\
\langle\!\langle ((\langle\rangle\langle l = a\rangle\langle m = b\rangle) \backslash (\langle\rangle\langle m = c\rangle)) \rightarrow\; m \rangle\!\rangle = \langle\rangle \\
\langle\!\langle (\langle\rangle\langle l = a\rangle\langle m = \langle\rangle\langle n = c\rangle\rangle) \rightarrow\; m \rangle\!\rangle = \langle\rangle\langle n = c\rangle \\
\langle\!\langle ((\langle\rangle\langle l = a\rangle\langle m = \langle\rangle\langle n = c\rangle\rangle) \cdot (\langle\rangle\langle m = d\rangle)) \rightarrow\; m \rangle\!\rangle = \langle\rangle
\end{array}
$$

In the form $\langle\rangle\langle l=a\rangle\langle m=b\rangle$ the label $m$ binds an abstract value. Therefore, dereference evaluation yields $\langle\rangle$. The second dereference evaluation also yields $\langle\rangle$ because $\langle\rangle\langle m=c\rangle$ restricts $\langle\rangle\langle l=a\rangle\langle m=b\rangle$ by hiding the binding involving label $m$. In the third example, label $m$ binds a nested form. Hence, the result of the dereference evaluation is $\langle\rangle\langle n=c\rangle$. Finally, since the polymorphic extension $\langle\rangle\langle l=b\rangle\langle m=d\rangle$ binds the right-most label $m$ to an abstract value, the result of the dereference evaluation of the whole form is $\langle\rangle$.

Now, we can define $[\![]\!]^F$. The application of $[\![]\!]^F$ to a given form $F$ yields a form value $\hat{F}$.

Definition 3. *Let $F \in \mathcal{F}$ be a form. The evaluation of a form $F$, written $[\![F]\!]^F$, yields a form value $\hat{F} \in \hat{\mathcal{F}}$ and is inductively defined as follows:*

$$\begin{array}{lcllcl}
[\![\langle\rangle]\!]^F & = & \langle\rangle & [\![F]\!]^V & = & [\![F]\!]^F \\
[\![F\langle l=V\rangle]\!]^F & = & [\![F]\!]^F\langle l=[\![V]\!]^V\rangle & [\![S]\!]^V & = & [\![S]\!]^S \\
[\![F \cdot G]\!]^F & = & [\![F]\!]^F \cdot [\![G]\!]^F & & & \\
[\![F \backslash G]\!]^F & = & [\![F]\!]^F \backslash [\![G]\!]^F & [\![\mathcal{E}]\!]^S & = & \mathcal{E} \\
[\![F \rightarrow\ l]\!]^F & = & \langle\!\langle [\![F]\!]^F \rightarrow\ l \rangle\!\rangle & [\![a]\!]^S & = & a \\
 & & & [\![F_l]\!]^S & = & [\![([\![F]\!]^F)_l]\!]
\end{array}$$

The function $[\![]\!]^F$, while preserving all polymorphic extensions and polymorphic restrictions, evaluates all projections and form dereferences in $F$ and replaces them by their corresponding value. In fact, $[\![F]\!]^F$ yields a form value $\hat{F}$ that does not contain any projection or dereference.

The effect of *form evaluation* is shown in the following example:

$$\begin{array}{cl}
 & [\![\langle\rangle\langle l=\langle\rangle\langle m=(\langle\rangle\langle k=\langle\rangle\langle m=w\rangle\rangle \rightarrow k)_m\rangle\rangle]\!]^F \\
= & [\![\langle\rangle]\!]^F\langle l=[\![\langle\rangle\langle m=(\langle\rangle\langle k=\langle\rangle\langle m=w\rangle\rangle \rightarrow k)_m\rangle]\!]^V\rangle \\
= & \langle\rangle\langle l=[\![\langle\rangle\langle m=(\langle\rangle\langle k=\langle\rangle\langle m=w\rangle\rangle \rightarrow k)_m\rangle]\!]^F\rangle \\
= & \langle\rangle\langle l=[\![\langle\rangle]\!]^F\langle m=[\![(\langle\rangle\langle k=\langle\rangle\langle m=w\rangle\rangle \rightarrow k)_m]\!]^V\rangle\rangle \\
= & \langle\rangle\langle l=[\![\langle\rangle]\!]^F\langle m=[\![(\langle\rangle\langle k=\langle\rangle\langle m=w\rangle\rangle \rightarrow k)_m]\!]^S\rangle\rangle \\
= & \langle\rangle\langle l=\langle\rangle\langle m=[\![([\![\langle\rangle\langle k=\langle\rangle\langle m=w\rangle\rangle \rightarrow k]\!]^F)_m]\!]\rangle\rangle \\
= & \langle\rangle\langle l=\langle\rangle\langle m=[\![(\langle\!\langle[\![\langle\rangle\langle k=\langle\rangle\langle m=w\rangle\rangle]\!]^F \rightarrow k\rangle\!\rangle)_m]\!]\rangle\rangle \\
= & \langle\rangle\langle l=\langle\rangle\langle m=[\![(\langle\!\langle[\![\langle\rangle]\!]^F\langle k=[\![\langle\rangle\langle m=w\rangle]\!]^V\rangle \rightarrow k\rangle\!\rangle)_m]\!]\rangle\rangle \\
= & \langle\rangle\langle l=\langle\rangle\langle m=[\![(\langle\!\langle\langle\rangle\langle k=[\![\langle\rangle\langle m=w\rangle]\!]^F\rangle \rightarrow k\rangle\!\rangle)_m]\!]\rangle\rangle \\
= & \langle\rangle\langle l=\langle\rangle\langle m=[\![(\langle\!\langle\langle\rangle\langle k=[\![\langle\rangle]\!]^F\langle m=[\![w]\!]^V\rangle\rangle \rightarrow k\rangle\!\rangle)_m]\!]\rangle\rangle \\
= & \langle\rangle\langle l=\langle\rangle\langle m=[\![(\langle\!\langle\langle\rangle\langle k=\langle\rangle\langle m=[\![w]\!]^S\rangle\rangle \rightarrow k\rangle\!\rangle)_m]\!]\rangle\rangle \\
= & \langle\rangle\langle l=\langle\rangle\langle m=[\![(\langle\!\langle\langle\rangle\langle k=\langle\rangle\langle m=w\rangle\rangle \rightarrow k\rangle\!\rangle)_m]\!]\rangle\rangle \\
= & \langle\rangle\langle l=\langle\rangle\langle m=[\![(\langle\rangle\langle m=w\rangle)_m]\!]\rangle\rangle \\
= & \langle\rangle\langle l=\langle\rangle\langle m=w\rangle\rangle \qquad \square
\end{array}$$

Even though a form may contain a binding for a label, say $l$, this form may be indistinguishable from a form that does not contain a binding for label $l$. In this case, we say that the label $l$ occurs transparent, which denotes the fact that a particular service or component interface is not available.

Definition 4. *A label $l$ is transparent with respect to form value $\hat{F}$, written $\hat{F} \parallel l$, iff*

$$[\![\hat{F}_l]\!] \ = \ \mathcal{E} \ \wedge \ [\![\hat{F} \rightarrow\ l]\!] \ = \ \langle\rangle.$$

Transparent labels can also be used for information hiding. For example, consider $\hat{F} \equiv \langle\rangle\langle l=a\rangle\langle k=(\langle\rangle \cdot \hat{G})\rangle$. We can hide the service located at label $l$ by applying the binding extension $\langle l=\mathcal{E}\rangle$, such that $\hat{F}' \equiv \hat{F}\langle l=\mathcal{E}\rangle$ and $\hat{F}' \parallel l$. Similarly, we can hide the component interface located at label $k$ with the binding extension $\langle k=\langle\rangle\rangle$, such that $\hat{F}'' \equiv \hat{F}\langle k=\langle\rangle\rangle$ and $\hat{F}'' \parallel k$. In both cases, the corresponding service and component interface, respectively, become inaccessible just as if the labels had never been defined.

The dual notion of $\hat{F} \parallel l$ is $\hat{F} \nparallel l$, which represents the fact that the value bound by label $l$ in a form value $\hat{F}$ is different from $\mathcal{E}$ and $\langle\rangle$, respectively. The property $\hat{F} \nparallel l$ does not state, however, which feature is actually provided by the form value $\hat{F}$; it can be either a service or a component interface.

Definition 5. *A label $l$ is nontransparent with respect to form value $\hat{F}$, written $\hat{F} \nparallel l$, iff*

$$[\![\hat{F}_l]\!] \ \neq \ \mathcal{E} \ \vee \ [\![\hat{F} \rightarrow\ l]\!] \ \neq \ \langle\rangle.$$

An important questions in our theory of forms is when two forms can be said to exhibit the same meaning. As in the $\lambda$-calculus, the most intuitive way of defining an equivalence of forms is via a notion of *contextual equivalence*.

A *form context* $\mathcal{C}[\cdot]$ is obtained when the hole $[\cdot]$ replaces an occurrence of a form (i.e., $F$) in the grammar of forms. We say that the forms $F$ and $G$ are equivalent, when $\mathcal{C}[F]$ and $\mathcal{C}[G]$ have the same "observable meaning" for each form context $\mathcal{C}[\cdot]$.

Definition 6. *Let $\hat{F}$ be a form value. Then the set of nontransparent labels of a form value $\hat{F}$, written $\hat{\mathcal{L}}_{\nparallel}(\hat{F})$, is defined as follows:*

$$\hat{\mathcal{L}}_{\nparallel}(\hat{F}) = \{\ l \in \mathcal{L} \mid \hat{F} \nparallel l \ \}$$

This definition gives rise to the definition of *behavioral equivalence* of forms.

Definition 7. *Two forms $F$ and $G$ are behaviorally equivalent, written $F \approx G$, if and only if, for all nontransparent labels $l \in \hat{\mathcal{L}}_{\nparallel}([\![F]\!]^F) \cup \hat{\mathcal{L}}_{\nparallel}([\![G]\!]^F)$,*

$$[\![([\![F]\!]^F)_l]\!] = [\![([\![G]\!]^F)_l]\!] \wedge (([\![F]\!]^F \rightarrow\ l) \approx ([\![G]\!]^F \rightarrow\ l))$$

.

Two forms $F$ and $G$ are equivalent if all projection evaluations of label $l \in \hat{\mathcal{L}}_{\nparallel}([\![F]\!]^F) \cup \hat{\mathcal{L}}_{\nparallel}([\![G]\!]^F)$ yield the same value for both forms and if all their nested forms bound by label $l$

are equivalent. In the case of $\hat{\mathcal{L}}_{\not\mathcal{T}}(\llbracket F \rrbracket^{F}) \cup \hat{\mathcal{L}}_{\not\mathcal{T}}(\llbracket G \rrbracket^{F}) = \emptyset$, two forms $F$ and $G$ both evaluate to $\langle\rangle$ and they are considered equivalent.

The relation defined by $\approx$ is an equivalence relation. Furthermore, $\approx$ is preserved by all form operations, that is, $(F \approx G) \Rightarrow (\mathcal{C}[F] \approx \mathcal{C}[G])$.

Rather than defining form equivalence over all labels in $\mathcal{L}$, we restrict the equivalence of two forms $F$ and $G$ to the union of their nontransparent labels. This is an optimization, but it can be shown that for all labels $l \notin \hat{\mathcal{L}}_{\not\mathcal{T}}(\llbracket F \rrbracket^{F}) \cup \hat{\mathcal{L}}_{\not\mathcal{T}}(\llbracket G \rrbracket^{F})$ both forms $F$ and $G$ exhibit also the same behavior.

Forms are immutable data structures. Over time, a form can grow, which results in the fact that much of its bindings become potentially inaccessible. Those bindings can be garbage collected. After garbage collection a so-called *normalized form* solely contains *binding extensions*.

We use $\bar{F}, \bar{G}, \bar{H}$ to range over the set $\bar{\mathcal{F}}$ of normalized form values. The set $\bar{\mathcal{F}}$ of normalized form values is a subset of $\hat{\mathcal{F}}$, i.e., $\bar{\mathcal{F}} \subset \hat{\mathcal{F}}$, and is defined as follows:

$$\bar{F} \quad ::= \quad \begin{cases} \langle\rangle & n = 0 \\ \langle\rangle\langle l_1 = v_1\rangle\langle l_2 = v_2\rangle \ldots \langle l_n = v_n\rangle & n > 0 \end{cases}$$

where

- all labels $l_i$ are pairwise distinct, that is, for all $i, j \in \{1, \ldots, n\}$ with $i \neq j$, it holds that $l_i \neq l_j$, and
- each value $v_i$ with $i \in \{1, \ldots, n\}$ is either an abstract value different from $\mathcal{E}$ or a non-empty normalized form.

With *normalized forms*, we recover classical records. However, we still maintain position independency, that is, it holds that $\langle\rangle\langle l = a\rangle\langle m = b\rangle \approx \langle\rangle\langle m = b\rangle\langle l = a\rangle$.

For every form $F$ there exists a normalized form $\bar{F}$, such that $F \approx \bar{F}$. This normalized form can be generated by the following algorithm:

**let**
  Normalize($\hat{F}$, $\emptyset$) = $\langle\rangle$
  Normalize($\hat{F}$, $\{l\} \cup \hat{\mathcal{L}}$) =
    **if** $\llbracket \hat{F}_l \rrbracket \neq \mathcal{E}$
      **then** (Normalize($\hat{F}$, $\hat{\mathcal{L}}$))$\langle l = \llbracket \hat{F}_l \rrbracket\rangle$
      **else**
        **let**
          $\hat{G}$ = Normalize($\langle\!\langle \hat{F} \rightarrow l \rangle\!\rangle, \hat{\mathcal{L}}(\langle\!\langle \hat{F} \rightarrow l \rangle\!\rangle)$)
        **in**
          **if** $\hat{G} \neq \langle\rangle$
            **then** (Normalize($\hat{F}$, $\hat{\mathcal{L}}$))$\langle l = \hat{G}\rangle$
            **else** Normalize($\hat{F}$, $\hat{\mathcal{L}}$)
**in**
  Normalize($\llbracket F \rrbracket^{F}$, $\hat{\mathcal{L}}(\hat{F})$)

Using this algorithm, we can always transform a given form $F$ into its corresponding behaviorally equivalent normalized form $\bar{F}$.

# 4. APPLICATION OF FORMS

Forms are used to represent both components and component interfaces. This in turn requires that forms are compile-time and run-time entities. As compile-time entities forms are used to specify component interfaces and component interface composition. At run-time, forms provide a uniform access to component services in an object-oriented way. In fact, being run-time entities forms may also allow for dynamic composition scheme or a hot-swap of components.

What is a software component? Using the characterization defined by Nierstrasz [17], a software component is a "static abstraction with plugs". But this characterization is rather vague. In this paper, we propose a new paradigm that is based on *module interconnection languages* [8] and *traits* [25].

In our new paradigm, a component is a collection of cooperating objects, each representing a partial state of the component. This approach stresses the view that, in general, we need a set of programming entities (or objects) to represent one component. We consider a one-to-one relationship between an object and a component (i.e., one object completely implements the semantics of a component) a special case. This is somewhat a departure from the approach mostly used today to represent components on top of an object-oriented programming language or system.

The provided and required services of a component are modeled by method pointers (or delegates) and forms are used to represent component interfaces. It is also possible to specify multiple forms (i.e., component interfaces) for the same collection of cooperating objects (i.e., the component), by allowing that different interfaces can share the same delegates. Using this approach, we can think of forms as traits with the exception that the services specified in forms have also an associated state. In fact, forms provide an abstraction similar to modules and the form operations can be used to combine modules.

In order to illustrate our approach, we will use some software artifacts developed in the .NET framework. In particular, we will show how these artifacts (i.e., C# code) can be encapsulated by forms using delegates as instantiations of abstract values. We can do so, because in the .NET framework both (structural) equality and inequality are defined for delegates.

Delegates are one of the most notable innovations of the C# language and the .NET framework. Delegates are type-safe, secure managed objects that behave like *method pointers* [3, 10, 29]. A delegate is a reference type that can encapsulate a method with a particular signature and a return type. In the .NET framework, delegates are mainly used to specify events and callbacks.

Delegates are defined using the `delegate` keyword, followed by a return type and a method signature. For example, consider the following delegate declarations:

```
public delegate void Setter( Object aValue );
public delegate Object Getter();
```

These declarations define the delegates `Setter` and `Getter`, which can encapsulate any method that takes an object as parameter and has the return type `void` (`Setter`), or has no parameters and returns an object (`Getter`).

However, while primarily being used for events and callbacks, delegates also provide a level of abstraction that enables us to use them to represent component plugs. In fact, a plug can be considered as callback that, when notified, provides a service or requires a service in turn.

Now, consider a generic *storage cell*, which represents an updatable data structure. A storage cell maintains a private state (i.e., its contents), and has at least two methods: `get` to read its contents and `set` to update its contents, respectively. A generic[1] C# implementation is shown in the following:

```
// generic reference cell
class StorageCell
{
  // Contents
  private Object fContents;

  // Getter method
  public Object get()
    { return fContents; }

  // Setter method
  public void set( Object aValue )
    { fContents = aValue; }
}
```

Now, this C# storage cell can be represented by the following form:

$$StorageCell \equiv \langle\rangle\langle get = \text{aObjGetter}\rangle\langle set = \text{aObjSetter}\rangle$$

In the form *StorageCell*, both `aObjGetter` and `aObjSetter` are delegates. The required C# code to define both delegates is shown in the following code fragment:

```
// create a fresh storage cell object
StorageCell aObj = new StorageCell();

// create the delegates
Setter aObjSetter = new Setter( aObj.set );
Getter aObjGetter = new Getter( aObj.get );
```

To set the contents of our storage cell component to the string value `''A new string value''`, we can use the following pseudo-code expression:

[1]The .NET framework has a fully object-oriented data model, where every data type is derived from `Object`. Therefore, we can use this data type to enable a storage cell to hold values of any .NET data type.

$$[\![([\![StorageCell]\!]^F)_{set}]\!](\text{ "A new string value" })$$

In this expression, $[\![([\![StorageCell]\!]^F)_{set}]\!]$ yields the delegate `aObjSetter`, which, when applied to the string argument, invokes `aObj`'s `set` method.

Similarly, to extract the current contents from our storage cell component, we can use

$$[\![([\![StorageCell]\!]^F)_{get}]\!]()$$

in which, $[\![([\![StorageCell]\!]^F)_{get}]\!]$ yields the `aObjGetter` delegate that, when called, invokes `aObj`'s `get` method.

In a second example, we illustrate the composition of two components using an approach similar to COM/ActiveX aggregation [23]. Suppose we have a `Multiselector` and a `GUIList` component. The `GUIList` component provides two services `paint` and `close` whereas the `Multiselector` provides the services `select`, `deselect`, and `close`. Both components can be represented by the following forms:

$$Multiselector \equiv \langle\rangle\langle select = \text{s}\rangle\langle deselect = \text{d}\rangle\langle close = \text{c}\rangle$$
$$GUIList \equiv \langle\rangle\langle paint = \text{p}\rangle\langle close = \text{c2}\rangle$$

where `s`, `d`, `c`, `p`, and `c2` are delegates. The required C# code to define the delegates is shown in the following code fragment:

```
// delegate type definition
public delegate void GuiOp();

// GUI objects
Multiselector aMultiselector =
                   new Multiselector();
GUIList aGUIList = new GUIList();

// plug delegates
GuiOp s = new GuiOp( aMultiselector.select );
GuiOp d = new GuiOp( aMultiselector.deselect );
GuiOp c = new GuiOp( aMultiselector.close );
GuiOp p = new GuiOp( aGUIList.paint );
GuiOp c2 = new GuiOp( aGUIList.close );
```

A composition of these two components has to offer the union of both sets of services, and, in order to close the composite component correctly, an invocation of `close` must be forwarded to both components. In order to define the required dispatch of `close`, we have to define some *glue code*. That is, we construct a new delegate `dispatchClose` and register the delegates yielded by the projection evaluations $[\![([\![Multiselector]\!]^F)_{close}]\!]$ and $[\![([\![GUIList]\!]^F)_{close}]\!]$:

`dispatchClose` $\equiv$
$$[\![([\![Multiselector]\!]^F)_{close}]\!] + [\![([\![GUIList]\!]^F)_{close}]\!];$$

In this glue pseudo-code, the delegate `dispatchClose` is actually a multicast delegate that, when called, invokes all registered delegates.

Using the newly defined delegate, we can define our first composite `fixedcompose`:

$$
\begin{aligned}
&fixedcompose \equiv \\
&\qquad \langle\rangle \\
&\qquad \langle select = [\![ ([\![ Multiselector ]\!]^F)_{select} ]\!] \rangle \\
&\qquad \langle deselect = [\![ ([\![ Multiselector ]\!]^F)_{deselect} ]\!] \rangle \\
&\qquad \langle paint = [\![ ([\![ GUIList ]\!]^F)_{paint} ]\!] \rangle \\
&\qquad \langle close = \text{dispatchClose} \rangle
\end{aligned}
$$

However, even though `fixedcompose` represents a composite component with the required functionality, it is not flexible enough. In fact, it can only work on instances like `Multiselector` and `GUIList`. If applied to components that provide additional services, then `fixcompose` will simply discard them.

To address this problem, we can define a new form, called `flexcompose`, that provides a generic abstraction, which, unlike `fixedcompose`, can work on instances that provide additional like `resize` or `selectall`.

$$
\begin{aligned}
&flexcompose \equiv \\
&\qquad (Multiselector \cdot GUIList)\langle close = \text{dispatchClose} \rangle
\end{aligned}
$$

The polymorphic extension $(Multiselector \cdot GUIList)$ defines an aggregation of the components `Mulitselector` and `GUIList`. Moreover, the use of polymorphic extension guaratees that even if `Multiselector` and `GUIList` provide additional services, `flexcompose` will not discard them.

The proper dispatch of `close` is guaranteed by the binding extension $\langle close = \text{dispatchClose} \rangle$. Since it is the right-most binding extension, it hides all bindings involving label `close` in $(Multiselector \cdot GUIList)$. Therefore, an application of the `close` service $[\![ ([\![ flexcompose ]\!]^F)_{close} ]\!]()$ will have the desired effect.

## 5. CONCLUSION AND FUTURE WORK

We have presented a small theory of forms, a special notion of immutable extensible records. Forms are the key concepts for extensibility, flexibility, and robustness in component-based application development. Furthermore, forms enable the definition of a canonical set compositional abstractions in an uniform framework.

In this paper, we have focused on the representation of components and component interfaces based on a notion of symbols. In fact, there exists a close relationship between forms and XML [31]. Both forms and XML can be used as platform-independent specifications of data types (e.g. component interfaces). In order to provide more expressiveness of forms, we are currently working of a notion of binding extension that also incorporate attribute specification that will enable us to specify additional functional and non-functional properties of services.

Forms are very similar to *traits* [25]. However, unlike traits forms incorporate the notion of state, if services are represented by delegates. On the other hand, both traits and forms do not affect the semantics of the underlying program entities and their composition mechanisms have similar effects.

Furthermore, we have shown that forms can be used to represent components written in C#. In fact, we have presented a new paradigm that characterizes components as collections of cooperating objects. The main idea of this approach is the use of delegates as component plugs. In fact, together with the notion of forms, delegates are most useful to define robust and reusable software abstractions.

However, forms do not exist in isolation. In fact, forms have to be embedded into a concrete computational model like the $\lambda$-calculus or the $\pi$-calculus. Then it will be possible to define true, parameterized compositional abstractions.

Future work will also include the definition of an appropriate typing scheme for forms, as types impose constraints which help to enforce the correctness of a program [5]. The plugs of a component constitute essentially an interface, or a *contractual specification*. Ideally, all conditions of a contract should be stated explicitly and formally as part of an interface specification. Furthermore, it would be highly desirable to have tools to check automatically clients and providers against the contractual specifications and in the case of a violation to reject the interaction of both.

The problem of inferring a contractual type for a given component $A$ can be stated as the problem of finding (i) a type $P$ that represents what component $A$ provides, (ii) a type $R$ that represents what component $A$ requires of a deployment environment, and (iii) a set of constraints $C$, which must be satisfied by provided type $P$ due to requirements posed by $R$. That is, whenever it is possible to infer (or prove the existence of) $P$, $R$, and $C$, software components can be safely composed.


## 6. ACKNOWLEDGMENTS

We would like to thank Gary Leavens for inspiring discussions on these topics as well as the anonymous reviewers for commenting on an earlier draft.


## 7. REFERENCES


[1] F. Achermann and O. Nierstrasz. Explicit Namespaces. In J. Gutknecht and W. Weck, editors, *Modular Programming Languages*, LNCS 1897, pages 77–99. Springer, Sept. 2000.

[2] R. J. Allen. *A Formal Approach to Software Architecture*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, May 1997.

[3] T. Archer. *Inside C#*. Mircosoft Press, 2001.

[4] L. Cardelli and J. C. Mitchell. Operations on Records. In C. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*. MIT Press, 1994. Also appeared as SRC Research Report 48, and in *Mathematical Structures in Computer Science*, 1(1):3–48, March 1991.

[5] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, Dec. 1985.

[6] L. Dami. *Software Composition: Towards an Integration of Functional and Object-Oriented Approaches*. PhD thesis, Centre Universitaire d'Informatique, University of Geneva, CH, 1994.

[7] L. Dami. A Lambda-Calculus for Dynamic Binding. *Theoretical Computer Science*, 192:201–231, Feb. 1998.

[8] F. DeRemer and H. H. Kron. Programming in the Large versus Programming in the Small. *IEEE Transactions on Software Engineering*, SE–2(2):80–86, June 1976.

[9] G. Leavens and M. Sitamaran, editors. *Foundations of Component-Based Systems*. Cambridge University Press, Mar. 2000.

[10] J. Liberty. *Programming C#*. O'Reilly, 2nd edition, 2002.

[11] M. Lumpe. *A $\pi$-Calculus Based Approach to Software Composition*. PhD thesis, University of Bern, Institute of Computer Science and Applied Mathematics, Jan. 1999.

[12] M. Lumpe, J.-G. Schneider, O. Nierstrasz, and F. Achermann. Towards a formal composition language. In G. T. Leavens and M. Sitaraman, editors, *Proceedings of ESEC '97 Workshop on Foundations of Component-Based Systems*, pages 178–187, Zurich, Sept. 1997.

[13] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In W. Schäfer and P. Botella, editors, *Proceedings ESEC '95*, LNCS 989, pages 137–153. Springer, Sept. 1995.

[14] Microsoft Corporation. *Visual Basic Programmierhandbuch*, 1997.

[15] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[16] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, Part I/II. *Information and Computation*, 100:1–77, 1992.

[17] O. Nierstrasz and L. Dami. Component-Oriented Software Technology. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 3–28. Prentice Hall, 1995.

[18] O. Nierstrasz and T. D. Meijler. Requirements for a Composition Language. In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, LNCS 924, pages 147–161. Springer, 1995.

[19] O. Nierstrasz and T. D. Meijler. Research directions in software composition. *ACM Computing Surveys*, 27(2):262–264, June 1995.

[20] M. Papathomas. A Unifying Framework for Process Calculus Semantics of Concurrent Object-Oriented Languages. In M. Tokoro, O. Nierstrasz, and P. Wegner, editors, *Proceedings of the ECOOP '91 Workshop on Object-Based Concurrent Computing*, LNCS 612, pages 53–79. Springer, 1992.

[21] M. Papathomas. Behaviour Compatibility and Specification for Active Objects. In D. Tsichritzis, editor, *Object Frameworks*, pages 31–40. Centre Universitaire d'Informatique, University of Geneva, July 1992.

[22] J. Richter. *Applied Microsoft .NET Framework Programming*. Mircosoft Press, 2002.

[23] D. Rogerson. *Inside COM: Microsoft's Component Object Model*. Microsoft Press, 1997.

[24] J. Sametinger. *Software Engineering with Reusable Components*. Springer, 1997.

[25] N. Schärli, D. Stéphane, O. Nierstrasz, and A. Black. Traits: Composable Units of Behavior. In L. Cardelli, editor, *Proceedings of the ECOOP '03*, LNCS 2743, pages 248–274. Springer, July 2003.

[26] J.-G. Schneider. *Components, Scripts, and Glue: A conceptual framework for software composition*. PhD thesis, University of Bern, Institute of Computer Science and Applied Mathematics, Oct. 1999.

[27] J.-G. Schneider and M. Lumpe. Synchronizing Concurrent Objects in the Pi-Calculus. In R. Ducournau and S. Garlatti, editors, *Proceedings of Langages et Modèles à Objets '97*, pages 61–76, Roscoff, Oct. 1997. Hermes.

[28] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[29] A. Troelsen. *C# and the .NET Platform*. Apress, 2001.

[30] G. van Rossum. Python Reference Manual. Technical report, Corporation for National Research Initiatives (CNRI), Oct. 1996.

[31] W3C Recommendation. *Extensible Markup Language (XML) 1.0 (Second Edition)*, Oct. 2000. http://www.w3.org/TR/REC-xml.

[32] L. Wall, T. Christiansen, and R. L. Schwartz. *Programming Perl*. O'Reilly & Associates, 2nd edition, Sept. 1996.

# Algorithmic Game Semantics and Component-Based Verification

Samson Abramsky, Dan R. Ghica, Andrzej S. Murawski, C.-H. Luke Ong
Oxford University Computing Laboratory

## ABSTRACT

We present a research programme dedicated to the application of Game Semantics to program analysis and verification. We highlight several recent theoretical results and describe a prototypical software modeling and verification tool. The distinctive novel features of the tool are its ability to handle *open programs* and the fact that the models it produces are *observationally fully abstract*. These features are essential in the modeling and verification of software components such as modules. Incidentally, these features also lead to very compact models of programs.

## 1. INTRODUCTION AND BACKGROUND

Game Semantics has emerged as a powerful paradigm for giving semantics to a variety of programming languages and logical systems. It has been used to construct the first syntax-independent fully abstract models for a spectrum of programming languages ranging from purely functional languages to languages with non-functional features such as control operators and locally-scoped references [4, 27, 5, 6, 3, 28].

We are currently developing Game Semantics in a new, algorithmic direction, with a view to applications in computer-assisted verification and program analysis. Some promising steps have already been taken in this direction. Hankin and Malacaria have applied Game Semantics to program analysis, e.g. to certifying secure information flows in programs [21, 22]. A particularly striking development was the work by Ghica and McCusker [20] which captures the game semantics of a procedural language in a remarkably simple form, as regular expressions. This leads to a decision procedure for observational equivalence on this fragment. Ghica has subsequently extended the approach to a call-by-value language with arrays [16], to model checking Hoare-style program correctness assertions [15] and to a more general model-checking friendly specification framework [17].

Game Semantics has several features which make it very promising from this point of view. It provides a very *concrete* way of building *fully abstract* models. It has a clear operational content, while admitting *compositional methods* in the style of denotational semantics. The basic objects studied in Game Semantics are games, and strategies on games. Strategies can be seen as certain kinds of highly-constrained processes, hence they admit the same kind of automata-theoretic representations central to model checking and allied methods in computer-assisted verification. Moreover, games and strategies naturally form themselves into rich mathematical structures which yield very accurate models of advanced high-level programming languages, as the various full abstraction results show. Thus the promise of this approach is to carry over the methods of model checking (see e.g. [10]), which has been so effective in the analysis of circuit designs and communications protocols, to much more *structured* programming situations, in which data-types as well as control flow are important.

A further benefit of the algorithmic approach is that by embodying game semantics in tools, and making it concrete and algorithmic, it should become more accessible and meaningful to practitioners. We see Game Semantics as having the potential to fill the role of a "Popular Formal Semantics," called for in an eloquent paper by Schmidt [39], which can help to bridge the gap between the semantics and programming language communities. Game Semantics has been successful in its own terms as a semantic theory; we aim to make it useful to and usable by a wider community.

Model checking for state machines is a well-studied problem (e.g. Mur$\phi$ [14], Spin [25] and Mocha [8] to name a few systems). Software model checking is a relatively new direction (see e.g. [24]); the leading projects (e.g. SLAM [9], and *Bandera* [12]) excel in tool constructions. The closest to ours in terms of target applications is the SLAM project, which is able to check safety properties of C programs. This task is reduced in stages to the problem of checking if a given statement in an instrumented version of the program in question is reachable, using ideas from data-flow and inter-procedural analysis and abstract interpretation.

In relation to the extensive current activity in model checking and computer assisted verification, our approach is distinctive, being founded on a highly-structured *compositional* semantic model. This means that we can directly apply our methods to *open program phrases* (i.e. terms-in-context with free variables) in a high-level language with procedures, local variables and data types. This ability is essential in analyzing properties of software components. The soundness of our methods is guaranteed by the properties of the semantic models on which they are based. By contrast, most current model checking applies to relatively "flat" unstructured situations.

Our semantics-driven approach has some other additional benefits: it is generic and fully automated. We do not target particular bugs or programs. The tool has the level of automation of a compiler. The input is a program fragment, with very little instrumentation required, and the output is a finite-state (FS) model. The resulting model itself can be analyzed using third-party model-checking

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.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

tools, or our tool can automatically extract traces with certain properties, e.g. error traces.

Software model checking is a fast-developing area of study, driven by needs of the industry as much as, if not more than, theoretical results. Often, tool development runs well ahead of rigorous considerations of soundness of the methods being developed. Our aim is to build on the tools and methods which have been developed in the verification community, while exploring the advantages offered by our semantics-directed approach.

# 2. A PROCEDURAL PROGRAMMING LANGUAGE

Our prototypical procedural language is a simply-typed call-by-name lambda calculus with basic types of booleans (**bool**), integers (**exp**), assignable variables (**var**) and commands (**comm**). We denote the basic types by $\sigma$ and the function types by $\theta$. *Assignable* variables, storing integers, form the state while commands change the state. In addition to abstraction ($\lambda x:\sigma.M$) and application ($FA$), other terms of the language are conditionals, uniformly applied to any type, ($\mathbf{if}\, B\, \mathbf{then}\, M\, \mathbf{else}\, N$), recursion ($\mathbf{fix}\, x:\sigma.M$), constants (integers, booleans) and arithmetic-logic operators ($M * N$); we also have command-type terms which are the standard imperative operators: dereferencing (explicit in the syntax, $!V$), assignment ($V := N$), sequencing ($C; M$, note that we allow, by sequencing, expressions with side-effects), no-op (**skip**) and local variable block ($\mathbf{new}\, x\, \mathbf{in}\, M$). We write $M:\sigma$ to indicate that term $M$ has type $\sigma$.

This language, which elegantly combines state-based procedural and higher-order functional programming, is due to Reynolds [38] and its semantic properties have been the object of important research [35].

If the programming language is restricted to first-order procedures, (more precisely, we restrict types to $\theta ::= \sigma \mid \sigma \to \theta$) tail recursion (iteration) and finite data-types then the Abramsky-McCusker fully abstract game model for this language [5] has a very simple and appealing regular-language representation [20]. The formulation of the regular-language model in loc. cit. is very well suited for proving equivalences "by hand," but we will prefer a slightly different but equivalent presentation [2] because it is more uniform and more compact. The referenced work gives motivation and numerous examples for the model presented below.

## 2.1 Abstract syntax

The typing judgements have the form $\Gamma \vdash M:\theta$ where $\Gamma = x_1:\theta_1, \ldots, x_k:\theta_k$. The typing rules are those of the typed $\lambda$-calculus: variables, abstraction and application:

$$\frac{}{\Gamma, x:\theta \vdash x:\theta} \qquad \frac{\Gamma, x:\theta \vdash M:\theta'}{\Gamma \vdash \lambda x:\theta.M:\theta \to \theta'}$$

$$\frac{\Gamma \vdash M:\theta \to \theta' \qquad \Gamma \vdash M':\theta}{\Gamma \vdash MM':\theta'}$$

Additionally, there is a rule for block structure:

$$\frac{\Gamma, x:\mathbf{var} \vdash M:\sigma}{\Gamma \vdash \mathbf{new}\, x\, \mathbf{in}\, M:\sigma}$$

The programming language also contains a set of constants:

$$n:\mathbf{exp} \quad \mathbf{true}:\mathbf{bool} \quad \mathbf{false}:\mathbf{bool} \quad \mathbf{skip}:\mathbf{comm}$$
$$- := - : \mathbf{var} \to \mathbf{exp} \to \mathbf{comm}$$
$$\mathbf{if}\ -\ \mathbf{then}\ -\ \mathbf{else}\ - : \mathbf{bool} \to \sigma \to \sigma \to \sigma$$
$$-;- : \mathbf{comm} \to \sigma \to \sigma$$
$$\mathbf{while}\ -\ \mathbf{do}\ - : \mathbf{bool} \to \mathbf{comm} \to \mathbf{comm}$$

For the purpose of defining the semantics, it is convenient to use a variant of the above system, in which the application rule is replaced by two rules: linear application and contraction.

$$\frac{\Gamma \vdash M:\theta \to \theta' \qquad \Gamma' \vdash M':\theta}{\Gamma, \Gamma' \vdash MM':\theta'}$$

$$\frac{\Gamma, x:\theta, x':\theta \vdash M:\theta'}{\Gamma, y:\theta \vdash M[y/x, y/x']:\theta'}$$

It is well known that this system has the same typing judgements as the original system.

We also use a construct for function (or procedure) definition:

$$\frac{\Gamma \vdash M:\theta \qquad \Gamma, f:\theta \vdash N:\sigma}{\Gamma \vdash \mathbf{let}\, f\, \mathbf{be}\, M\, \mathbf{in}\, N:\sigma}$$

Finally, it is convenient to define a non-terminating command **div** : **comm**.

## 2.2 Extended Regular Expressions

This section describes the representation of the game model using a language of extended regular expressions. Due to space constraints, a basic understanding of game semantics must be assumed as background. Otherwise, the reader is encouraged to refer to the literature mentioned in the Introduction.

Terms are interpreted by languages over alphabets of moves $\mathcal{A}$. The languages, denoted by $\mathcal{L}(R)$, are specified using extended regular expressions $R$. They include the standard regular expressions consisting of the empty language $\emptyset$, the empty sequence $\epsilon$, concatenation $R \cdot S$, union $R + S$, Kleene star $R^*$, and the elements of the alphabet taken as sequences of unit length. We also use the additional constructs of intersection $R \cap S$, direct image under homomorphism $\phi R$ and inverse image $\phi^{-1} R$. The languages defined by these extensions are the obvious ones:

$$\mathcal{L}(R \cap S) = \mathcal{L}(R) \cap \mathcal{L}(S)$$
$$\mathcal{L}(\phi R) = \{\phi w \mid w \in \mathcal{L}(R)\}$$
$$\mathcal{L}(\phi^{-1} R) = \{w \in \mathcal{A}_1^* \mid \phi w \in \mathcal{L}(R)\},$$

where $\phi:\mathcal{A}_1 \to \mathcal{A}_2^*$ is a homomorphism; it lifts to strings in the usual way, $\phi(a_1 \ldots a_k) = \phi(a_1) \cdots \phi(a_k)$.

It is a standard result that any extended regular expression constructed from the operations described above denotes a regular language, which can be recognized by a finite automaton which can be effectively constructed from the regular expression [26].

We will often use the disjoint union of two alphabets to create a larger alphabet:

$$\mathcal{A}_1 + \mathcal{A}_2 = \{a^{\langle 1 \rangle} \mid a \in \mathcal{A}_1\} \cup \{b^{\langle 2 \rangle} \mid b \in \mathcal{A}_2\} = \mathcal{A}_1^{\langle 1 \rangle} \cup \mathcal{A}_2^{\langle 2 \rangle}.$$

The tags $-^{\langle i \rangle}$ are used on a lexical level, resulting in new and distinct symbols belonging to the larger alphabet. The disjoint union gives rise to the canonical maps:

$$\mathcal{A}_1 \underset{\text{outl}}{\overset{\text{inl}}{\rightleftarrows}} \mathcal{A}_1 + \mathcal{A}_2 \underset{\text{outr}}{\overset{\text{inr}}{\leftrightarrows}} \mathcal{A}_2$$

The definition of the maps is:

$$\text{inl}\, a = a^{\langle 1 \rangle} \qquad \text{inr}\, b = b^{\langle 2 \rangle}$$
$$\text{outl}\, a^{\langle 1 \rangle} = a \qquad \text{outr}\, a^{\langle 1 \rangle} = \epsilon$$
$$\text{outl}\, b^{\langle 2 \rangle} = \epsilon \qquad \text{outr}\, b^{\langle 2 \rangle} = b$$

If $\phi : \mathcal{A} \to \mathcal{B}^*$ and $\phi' : \mathcal{C} \to \mathcal{D}^*$ are homomorphisms then we define their sum $\phi + \phi' : \mathcal{A} + \mathcal{C} \to (\mathcal{B} + \mathcal{D})^*$ as

$$(\phi + \phi')(a^{\langle 1 \rangle}) = (\phi a)^{\langle 1 \rangle}$$
$$(\phi + \phi')(c^{\langle 2 \rangle}) = (\phi' c)^{\langle 2 \rangle}.$$

DEFINITION 1 (COMPOSITION). *If $R$ is a regular expression over alphabet $\mathcal{A} + \mathcal{B}$ and $S$ a regular expression over alphabet $\mathcal{B} + \mathcal{C}$ we define the* composition $R \circ S$ *as a regular expression over alphabet $\mathcal{A} + \mathcal{C}$*

$$R \circ S = out\big(out_1^{-1}(R) \cap out_2^{-1}(S)\big),$$

*with canonical maps*

$$\begin{array}{ccccc} \mathcal{A} + \mathcal{B} & \underset{out_1}{\overset{in_1}{\rightleftarrows}} & \mathcal{A} + \mathcal{B} + \mathcal{C} & \underset{out_2}{\overset{in_2}{\leftrightarrows}} & \mathcal{B} + \mathcal{C} \\ & & in \uparrow\downarrow out & & \\ & & \mathcal{A} + \mathcal{C} & & \end{array}$$

Regular expression composition is very similar to composition of finite state transducers [37]. Sets $\mathcal{A}$ and $\mathcal{B}$ represent, respectively, the input and the output of the first transducer; sets $\mathcal{B}$ and $\mathcal{C}$ represent, respectively, the input and the output of the second transducer. The result is a transducer of inputs $\mathcal{A}$ and output $\mathcal{C}$. For example, let $\mathcal{A} = \{a\}, \mathcal{B} = \{b\}, \mathcal{C} = \{c\}$; then $(ab)^* \circ (bcc)^* = (acc)^*$.

## 2.3 Alphabets

We interpret each type $\theta$ by a language over an alphabet $\mathcal{A}[\![\theta]\!]$, containing the *moves* from the game model. For basic types $\sigma$ it is helpful to define alphabets of questions $\mathcal{Q}[\![\sigma]\!]$ and answers $\mathcal{A}_q[\![\sigma]\!]$ for each $q \in \mathcal{Q}[\![\sigma]\!]$. The alphabet of type $\sigma$ is then defined as

$$\mathcal{A}[\![\sigma]\!] = \mathcal{Q}[\![\sigma]\!] \cup \bigcup_{q \in \mathcal{Q}[\![\sigma]\!]} \mathcal{A}_q[\![\sigma]\!].$$

The basic type alphabets are:

$$\begin{aligned} &\mathcal{Q}[\![\mathbf{exp}]\!] = \{\mathsf{q}\},\ \mathcal{A}_{\mathsf{q}}[\![\mathbf{exp}]\!] = \mathsf{N} \\ &\mathcal{Q}[\![\mathbf{bool}]\!] = \{\mathsf{q}\},\ \mathcal{A}_{\mathsf{q}}[\![\mathbf{bool}]\!] = \{\mathsf{t}, \mathsf{f}\} \\ &\mathcal{Q}[\![\mathbf{comm}]\!] = \{\mathsf{q}\},\ \mathcal{A}_{\mathsf{q}}[\![\mathbf{comm}]\!] = \{\star\} \\ &\mathcal{Q}[\![\mathbf{var}]\!] = \{\mathsf{q}\} \cup \{\mathsf{w(n)} \mid \mathsf{n} \in \mathsf{N}\}, \\ &\qquad \mathcal{A}_{\mathsf{q}}[\![\mathbf{var}]\!] = \mathsf{N}, \mathcal{A}_{\mathsf{w(n)}} = \{\star\}. \end{aligned}$$

where $\mathsf{N} = \{-n, \cdots, -1, 0, 1, \cdots, n\}$.

Alphabets of function types are defined by

$$\mathcal{A}[\![\sigma \to \theta]\!] = \mathcal{A}[\![\sigma]\!] + \mathcal{A}[\![\theta]\!].$$

A typing judgement $\Gamma \vdash M : \theta$ is interpreted by a regular expression $R = [\![\Gamma \vdash M : \theta]\!]$ over alphabet $\sum_{x_i : \theta_i \in \Gamma} \mathcal{A}[\![\theta_i]\!] + \mathcal{A}[\![\theta]\!]$.

For any type $\theta = \sigma_1 \to \cdots \to \sigma_k \to \sigma$, it is convenient to define a regular language $K_\theta$ over alphabet $\mathcal{A}[\![\theta]\!] + \mathcal{A}[\![\theta]\!]$, called the *copy-cat* language:

$$K_\theta = \sum_{q \in \mathcal{Q}[\![\sigma]\!]} q^{\langle 2 \rangle} \cdot q^{\langle 1 \rangle} \cdot \Big(\sum_{i=1,k} R_i\Big)^* \cdot \sum_{a \in \mathcal{A}_q[\![\sigma]\!]} a^{\langle 1 \rangle} \cdot a^{\langle 2 \rangle},$$

where

$$R_i = \sum_{q \in \mathcal{Q}[\![\sigma_i]\!]} q^{\langle 2 \rangle} \cdot q^{\langle 1 \rangle} \cdot \sum_{a \in \mathcal{A}_q[\![\sigma_i]\!]} a^{\langle 1 \rangle} \cdot a^{\langle 2 \rangle}.$$

This regular expression represents the so-called copy-cat strategy of game semantics, and it describes the generic behaviour of a sequential procedure. At second-order [36] and above [27] this behaviour is far more complicated.

## 2.4 Regular-language semantics

We interpret terms using an evaluation function $[\![-]\!]$ mapping a term $\Gamma \vdash M : \theta$ and an environment $u$ into a regular language $R$. The environment is a function, with the same domain as $\Gamma$, mapping identifiers of type $\theta$ to regular languages over $\mathcal{A}[\![\Gamma]\!] + \mathcal{A}[\![\theta]\!]$.

The evaluation function is defined by recursion on the syntax.

**Identifiers.** Identifiers are read from the environment:

$$[\![\Gamma, x : \theta \vdash x : \theta]\!] u = u(x).$$

**Abstraction.**

$$\begin{aligned} &[\![\Gamma \vdash \lambda x : \sigma . M : \sigma \to \theta]\!] u \\ &\qquad = \phi\big([\![\Gamma, x : \sigma \vdash M : \theta]\!](u \mid x \mapsto K_\sigma)\big) \end{aligned}$$

where $\phi$ is the (trivial) associative isomorphism

$$\phi : (\mathcal{A}[\![\Gamma]\!] + \mathcal{A}[\![\sigma]\!]) + \mathcal{A}[\![\theta]\!] \xrightarrow{\simeq} \mathcal{A}[\![\Gamma]\!] + (\mathcal{A}[\![\sigma]\!] + \mathcal{A}[\![\theta]\!]).$$

**Application and contraction.**

$$[\![\Gamma, \Delta \vdash MN]\!] u = [\![\Gamma \vdash M]\!] u \circ \big([\![\Delta \vdash N]\!] u\big)^*,$$

with composition $- \circ -$ defined as before. Contraction is

$$\begin{aligned} &[\![\Gamma, z : \theta \vdash M[z/x, z/x'] : \theta]\!] u \\ &\qquad = (\mathrm{id}_1 + \delta + \mathrm{id}_2)\big([\![\Gamma, x : \theta, x' : \theta \vdash M : \theta]\!] u\big), \end{aligned}$$

where $\mathrm{id}_1$ and $\mathrm{id}_2$ are identities on $\mathcal{A}[\![\Gamma]\!]$ and, respectively, $\mathcal{A}[\![\theta]\!]$. The homomorphism $\delta : \mathcal{A}[\![\theta]\!] + \mathcal{A}[\![\theta]\!] \to \mathcal{A}[\![\theta]\!]$ only removes tags from moves. Note that this interpretation is also specific to first-order types. In higher-order types this interpretation of contraction by un-tagging can result in ambiguities.

**Block Variables.** Consider the following regular expression over alphabet $\mathcal{A}[\![\mathbf{var}]\!]$

$$\text{cell} = \Big(\sum_{\mathsf{n} \in \mathsf{N}} \mathsf{w(n)} \cdot \star \cdot (\mathsf{q} \cdot \mathsf{n})^*\Big)^*.$$

Intuitively, one can see that this regular expression describes the sequential behaviour of a memory cell: if a value $n$ is written, then the same value is read back until the next write, and so on.

We define block variables as

$$[\![\Gamma \vdash \mathbf{new}\, x\, \mathbf{in}\, M : \sigma]\!] u = [\![\Gamma, x : \mathbf{var} \vdash M : \sigma]\!] u \circ \text{cell},$$

**Constants.** Finally, the interpretation of constants is:

$$\begin{aligned} &[\![n : \mathbf{exp}]\!] = \mathsf{q} \cdot \mathsf{n},\ [\![\mathbf{true} : \mathbf{bool}]\!] = \mathsf{q} \cdot \mathsf{t},\ [\![\mathbf{false} : \mathbf{bool}]\!] = \mathsf{q} \cdot \mathsf{f} \\ &[\![-\mathbf{op}- : \sigma \to \sigma \to \sigma']\!] \\ &\qquad = \sum_{\mathsf{p} \in \mathsf{N}} \sum_{\substack{\mathsf{m}, \mathsf{n} \in \mathsf{N} \\ \mathsf{p} = \mathsf{m}\, op\, \mathsf{n}}} \mathsf{q}^{\langle 3 \rangle} \cdot \mathsf{q}^{\langle 1 \rangle} \cdot \mathsf{m}^{\langle 1 \rangle} \cdot \mathsf{q}^{\langle 2 \rangle} \cdot \mathsf{n}^{\langle 2 \rangle} \cdot \mathsf{p}^{\langle 3 \rangle} \\ &[\![- := - : \mathbf{var} \to \mathbf{exp} \to \mathbf{comm}]\!] \\ &\qquad = \sum_{\mathsf{n} \in \mathsf{N}} \mathsf{q}^{\langle 3 \rangle} \cdot \mathsf{q}^{\langle 2 \rangle} \cdot \mathsf{n}^{\langle 2 \rangle} \cdot \mathsf{w(n)}^{\langle 1 \rangle} \cdot \star^{\langle 1 \rangle} \cdot \star^{\langle 3 \rangle} \\ &[\![\mathbf{if}\ -\ \mathbf{then}\ -\ \mathbf{else}\ - : \mathbf{bool} \to \sigma \to \sigma \to \sigma]\!] \\ &\qquad = \sum_{q \in \mathcal{Q}[\![\sigma]\!]} q^{\langle 4 \rangle} \cdot \mathsf{q}^{\langle 1 \rangle} \cdot \mathsf{t}^{\langle 1 \rangle} \cdot q^{\langle 2 \rangle} \cdot \sum_{a \in \mathcal{A}_q[\![\sigma]\!]} a^{\langle 2 \rangle} \cdot a^{\langle 4 \rangle} \\ &\qquad + \sum_{q \in \mathcal{Q}[\![\sigma]\!]} q^{\langle 4 \rangle} \cdot \mathsf{q}^{\langle 1 \rangle} \cdot \mathsf{f}^{\langle 1 \rangle} \cdot q^{\langle 3 \rangle} \cdot \sum_{a \in \mathcal{A}_q[\![\sigma]\!]} a^{\langle 3 \rangle} \cdot a^{\langle 4 \rangle} \end{aligned}$$

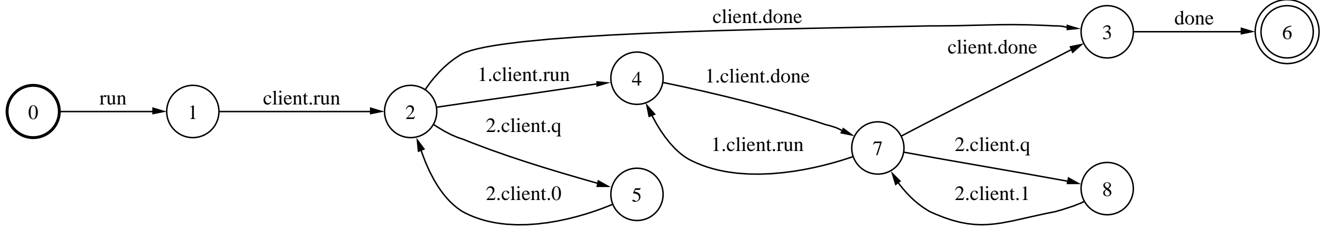



**Figure 1: A simple switch**

$$[\![-;-:\mathbf{comm}\to\sigma\to\sigma]\!] = \sum_{q\in\mathcal{Q}[\![\sigma]\!]} q^{\langle 3\rangle}\cdot \mathsf{q}^{\langle 1\rangle}\cdot\star^{\langle 1\rangle}\cdot q^{\langle 2\rangle}\cdot \sum_{a\in\mathcal{A}_q[\![\sigma]\!]} a^{\langle 2\rangle}\cdot a^{\langle 3\rangle}$$

$$[\![\mathbf{while}\ -\ \mathbf{do}\ -:\mathbf{bool}\to\mathbf{comm}\to\mathbf{comm}]\!] = \mathsf{q}^{\langle 3\rangle}\cdot\left(\mathsf{q}^{\langle 1\rangle}\cdot\mathsf{t}^{\langle 1\rangle}\cdot\mathsf{q}^{\langle 2\rangle}\cdot\star^{\langle 2\rangle}\right)^{*}\cdot\mathsf{q}^{\langle 1\rangle}\cdot\mathsf{f}^{\langle 1\rangle}\cdot\star^{\langle 3\rangle}$$

$$[\![\mathbf{div}:\mathbf{comm}]\!]=\emptyset,\qquad [\![\,\mathbf{skip}:\mathbf{comm}]\!]=\mathsf{q}\cdot\star.$$

The operator **op** ranges over the usual arithmetic-logic operators, and *op* is its obvious interpretation.

## 2.5 A warm-up example

This simple example illustrates quite well the way the game-based model works. It is a toy abstract data type (ADT): a switch that can be flicked on, with implementation:

```
client : com -> exp -> com |-
    new var v:= 0 in
    let set be v := 1 in
    let get be !v in
    client (set, get) : com.
```

The code consists of local integer variable `v`, storing the state of the switch, together with functions `set`, to flick the switch on, and `get`, to get the state of the switch. The initial state of the switch is *off*. The non-local, undefined, identifier `client` is declared at the left of the turnstile `|-`. It takes a command and an expression-returning functions as arguments. It represents, intuitively, "the most general context" in which this ADT can be used.

A key observation about the model is that the *internal state* of the program is abstracted away, and only the observable actions, of the *nonlocal* entity `client`, are represented, insofar as they contribute to terminating computations. The output of the modeling tool is given in Fig. 1.

Notice that no references to `v`, `set`, or `get` appear in the model! The model is only that of the possible behaviours of the `client`: whenever the client is executed, if it evaluates its second argument (get the state of the switch) it will receive the value 0 as a result; if it evaluates the first argument (set the switch on), one or more times, then the second argument (get the state of the switch) will always evaluate to 1. The model does not, however, assume that `client` uses its arguments, or how many times or in what order.

## 2.6 Full abstraction

Full abstraction results are crucial in semantics, as they are a strong qualitative measure of the semantic model. Full abstraction is defined with respect to observational equivalence: two terms are equivalent if and only if they can be substituted in all program contexts without any observable difference. This choice of observable is therefore canonical, and arises naturally from the programming language itself. In practice, fully abstract models are important because they identify all and only those programs which are observationally equivalent.

Formally, terms $M$ and $N$ are defined to be observationally equivalent, written $M \equiv N$, if and only if for any context $\mathcal{C}[-]$ such that both $\mathcal{C}[M]$ and $\mathcal{C}[N]$ are closed terms of type **comm**, $\mathcal{C}[M]$ converges if and only if $\mathcal{C}[N]$ converges. The theory of observational equivalence, which is very rich (see e.g. [20] for a discussion), has been the object of much research [35].

THEOREM 1 (FULL ABSTRACTION [5, 20]).

$$\Gamma\vdash M\equiv N \iff \mathcal{L}\big([\![\Gamma\vdash M:\theta]\!]u_0\big)=\mathcal{L}\big([\![\Gamma\vdash N:\theta]\!]u_0\big),$$

*where* $u_0(x)=K_\theta$ *for all* $x:\theta$ *in* $\Gamma$.

As an immediate consequence, observational equivalence for the finitary fragment discussed here is decidable.

It can be shown that the full abstraction result holds relative to contexts drawn from either the restricted fragment or the full programming language [19].

# 3. APPLICATIONS TO ANALYSIS AND VERIFICATION

The game model is *algorithmic, fully abstract* and *compositional*, therefore it provides excellent support for compositional program analysis and verification.

The initial decidability result of the previous section was extended to higher-order (recursion and iteration-free) call-by-name procedural programming by Ong [36] and, for call-by-value, by Murawski [34]. This required the use of deterministic pushdown automata [40, 41], since the associated sets of complete plays in the game semantics are no longer regular. Various other extensions of the programming fragment, e.g. by introducing unrestricted recursion [36] or further increasing the order of the fragment [33], lead to undecidability. The game-theoretic approach seems to offer a useful and powerful tool for investigating the algorithmic properties of programming language fragments, e.g. the complexity of program equivalence [32].

A different direction of research is the development of game-based, model-checking friendly specification languages. Such specification languages are necessary in order to fully exploit the com-

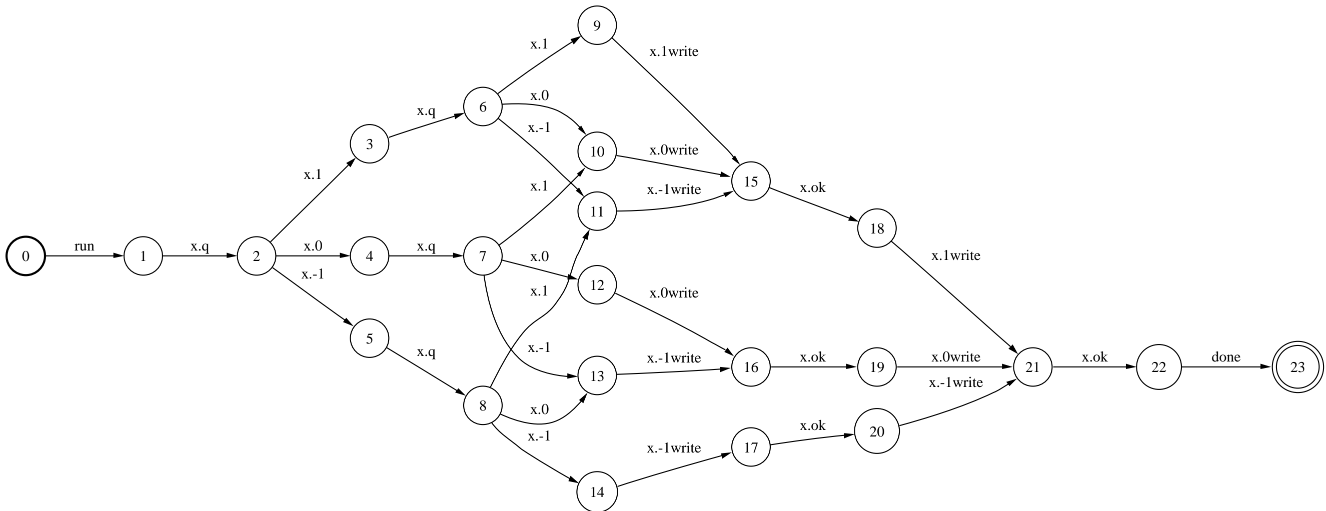



**Figure 2: A model of sorting**

positionality of the game-based approach. It is of little use to reason about program fragments if properties of the whole program cannot be then compositionally inferred, without requiring further model-checking. The first steps in this direction are taken in [17].

## 3.1 Tool support and case studies

The theoretical applications of game semantics have been very successful. However, since the complexity of the regular-language algorithms involved in the generation of the finite-state machines representing the game models is exponential (both in time and in space), it was unclear whether the technique was practicable. This is in fact a common situation in software model checking: the asymptotic complexity of the algorithms involved is high, but it turns out that the worst-case scenario only happens in pathological cases. Many programs can be in fact verified. But the only way to make such pragmatic assessments is to implement and experiment. We have implemented a prototype tool, and the results are very positive.

Our tool converts an open procedural program into the finite-state machine representation of the regular-language game model. Very little user instrumentation of the source code is required. The data-abstraction schemes (i.e. what finite sets of integers will be used to model integer variables) for integer-typed variables need to be supplied, using simple code annotations. The tool is implemented in CAML; most of the back-end heavy duty finite-state machine processing is done using the AT&T FSM library [1]. A more complete description of the tool is available in [18].

In the following we will present two case studies which best illustrate the distinctive features of our model: a sorting program and an abstract data type implementation.

## 3.2 Sorting

In this section we will discuss the modeling of a sorting program, a notoriously difficult problem. We will focus on *bubble-sort*, not for its algorithmic virtues but because it is one of the most straightforward non-recursive sorting algorithms. The implementation we

```
x:var |-                                                    1
  array a[n] in                                             2
  new var i:=0 in                                           3
  while !i < n do a[!i]:=!x; i:=!i+1 od;                    4
  new var flag:=1 in                                        5
  while !flag do                                            6
    new var i:=0 in                                         7
    flag:=0;                                                8
    while !i < n - 1 do                                     9
      if !a[!i] > !a[!i+1] then                             10
        flag:=1;                                            11
        new var temp:=!a[!i] in                             12
          a[!i]:=!a[!i+1];                                  13
          a[!i+1]:=!temp                                    14
        else skip fi;                                       15
      i:=!i+1                                               16
    od                                                      17
  od;                                                       18
  new var i:=0 in                                           19
  while !i < n do x:=!a[!i]; i:=!i+1 od : com.              20
```

**Figure 3: An implementation of sorting**

will analyze is the one in Fig. 3. Meta-variable $n$, representing the size of the array, will be instantiated to several different values. Observe that the program communicates with its environment using non-local **var**-typed identifier `x:var` only. Therefore, the model will only represent the actions of `x`. Since we are in a call-by-name setting, `x` can represent any **var**-typed procedure, for example interfacing with an input/output channel. Notice that the array being effectively sorted, `a[]`, is not visible from the outside of the program because it is locally defined.

We first generate the model for $n = 2$, i.e. an array of only

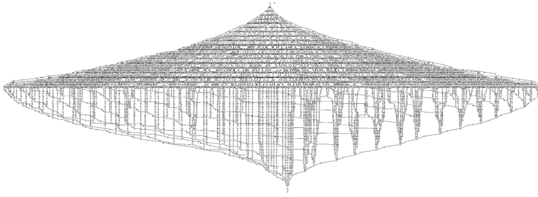


**Figure 4: A model of sorting: 20 element-array**

2 elements, in order to generate a small enough model which we can display and discuss. The type of stored data is integers in the interval $[-1, 1]$, i.e. 3 distinct values. The resulting model is as in Fig. 2. It reflects the dynamic behaviour of the program in the following way: every trace in the model is formed from the actions of reading all $3 \times 3 = 9$ possible combinations of values from x, followed by writing out the same values, but in sorted order.

Increases in the array lead to (asymptotically exponential) increases in the time and space of the verification algorithm. On our development machine (SunBlade 100, 2GB RAM), the duration of the generation of the model as a function of $n$ was: $n = 2$: 5 minutes; $n = 5$: 10 minutes; $n = 10$: 15 minutes; $n = 20$: 4 hours; $n = 25$: 10 hours; $n = 30$: the computation failed. Fig. 4 gives a snapshot of the model for $n = 20$.

The output is a FS machine, which can be analyzed using standard FS-based model checking tools. Moreover, this model is an *extensional* model of sorting: all sorting programs on an array of size $n$ will have isomorphic models. Therefore, a straightforward method of verification is to compare the model of a sorting program with the model of another implementation which is known to be correct. In the case of our finite-state models, this is a decidable operation.

Something quite remarkable about the model in Fig. 4 is its very compact size. An array of 20 (3-valued) elements can represent $3^{20}$ distinct states, i.e. approximately 3.5 billion states. This is a vast memory space, beyond the range of tools much more sophisticated than ours. Our tool cannot only handle such a program, but it also produces its *complete* model.

The key observation is the following: the fact that the state of the array is *internalized* and only a purely behavioural, observationally fully abstract model is presented leads to significant savings in required memory space. In fact, the model in Fig. 4 has only circa 6,500 states. So, even though the algorithms we use are generic, the fact that we use a model at a maximum level of abstraction, which internalizes the details of stateful behaviour leads to major improvements in efficiency. It is interesting to contrast this kind of abstraction, which comes for free with our fully abstract model, with other, syntactic, abstraction techniques such as slicing [23].

## 3.3 Code-level safety specifications

We define an assertion as a function which takes as argument a boolean, the condition to be asserted. It does nothing if the condition is true and calls an (nonlocal) `error` procedure if the condition is false. In the resulting model, any trace containing the actions `error.run, error.done` will represent a usage of the ADT which violates the invariant, i.e. an *error trace*.

The encoding of safety properties using code-level assertions is quite standard in SMC, e.g. [9], and it is also known that every safety property can be encoded in a regular language [31]. Using the assertion mechanism in conjunction with modeling open programs, such as modules, offers an elegant solution to the problem of checking equational properties or invariants of ADTs.

For example, consider an implementation of a finite-size stack, using a fixed-size array. The interface of the stack is through functions `push(n)` and `pop`. Their implementation is the obvious one (see Fig. 5). In addition, the stack component assumes the existence of functions `overflow` and `empty` to call if a `push` is attempted on a full stack, respectively a `pop` is attempted on an empty stack. These functions need not be implemented.

Suppose that we want to check, for a size 2 stack, whether it is the case that the last value pushed onto the stack is the value at the top of the stack. We do this by using the assertion `invariant` on lines 21–24 of Fig. 5. Notice the undefined component `VERIFY` of this program: it stands for *all* possible uses of the stack module and the assertion to be checked. The idea of providing such a generic closure of an open program can be traced back to [11], and several game-like solutions have been already proposed [13, 7]. The game model which we use provides this closure, correct and complete, directly at the level of the concrete programming language.

```
empty:com, overflow:com, m:exp, error:com,       1
VERIFY : com -> exp -> com -> com |-             2
      let assert be fun a : exp.                 3
          if a then skip else error fi in        4
      array buffer[n] in                         5
      let size be n in                           6
      new var crt:=0 in                          7
      let isempty be !crt = 0 in                 8
      let isfull be !crt = size in               9
      let push be fun x : exp.                   10
          new var temp:=x in                     11
          if isfull then overflow                12
          else buffer[!crt]:=!temp;              13
          crt:=!crt+1 fi                         14
          in                                     15
      let pop be                                 16
          if isempty then empty; 0               17
          else crt:=!crt - 1;                    18
               !buffer[!crt] fi                  19
          in                                     20
      let invariant be                           21
          new var x:=m in                        22
          push(!x); pop = !x                     23
          in                                     24
      VERIFY(push(m), pop, assert(invariant))    25
: com.                                           26
```

**Figure 5: A stack module**

The tool automatically builds the model for the above and extracts its shortest failure trace (see Fig. 6).

Action `1.VERIFY` represents a `push` action. So the simplest possible error is caused by pushing 3 times the value 1 onto the 2-element stack. Indeed, if the stack is already full, pushing a new element will cause an overflow error.

# 4. CURRENT LIMITATIONS AND FURTHER RESEARCH

The initial results of our effort to model and verify programs us-

```
0        1        run
1        2        VERIFY.run
2        3        1.VERIFY.run
3        4        m.q
4        5        m.1
5        6        1.VERIFY.done
6        7        1.VERIFY.run
7        8        m.q
8        9        m.1
9        10       1.VERIFY.done
10       11       3.VERIFY.run
11       12       m.q
12       13       m.0
13       14       overflow.run
14       15       overflow.done
15       16       error.run
16       17       error.done
17       18       3.VERIFY.done
18       19       VERIFY.done
19       20       done
20
```

**Figure 6: Shortest failure trace of stack component**

ing Game Semantics are very encouraging: this approach proves to give compact, practicable representations of many common programs, while the ability to model open programs allows us to verify software components, such as ADT implementations.

We are considering several further directions:

**language extensions:** the procedural language fragment we are currently handling only includes basic imperative and functional features. We are considering several ways to extend it: richer computational primitives such as concurrency and control, which already have game semantic models; restricted recursion schemes which are more expressive than iteration (i.e. tail recursion); higher-order functional features. In addition, we consider a version of this tool which would handle call-by-value languages.

**specifications:** in order to truly support compositional verification we intend to expand the tool to model *specifications* of open programs, rather than just open programs. A theoretical basis for that is already provided in [17], which is in turn inspired by the game-like ideas of *interface automata* [13].

**tools and methodology:** enriching the features of the tool and making it more robust and user friendly. For example, the definability result in [5] guarantees that any trace in the model can be mapped back into a program. Using this, we can give the user *code* rather than *trace* counterexamples to failed assertions. We would also like to investigate applying the tool to the modeling and verification of a larger, more realistic case study.

**scalable model checking:** our methods so far apply only to *finite* data and store. Verifying a program operating on finite data and store is an excellent method for bug detection and provides a fairly high measure of confidence in the correctness of the code, but it does not represent a *proof*. There is, in general, no guarantee that the properties of a program of given size generalize. But we hope that recent results in *data independence* [30, 29] can help overcome such limitations.

We are actively engaged in investigating the above topics, and we are grateful to the Engineering and Physical Sciences Research Council of the United Kingdom for financial support in the form of the research grant *Algorithmic Game Semantics and its Applications*; there is also a related project on *Scalable Software Model Checking based on Game Semantics* by Ranko Lazic of the University of Warwick.

# 5. REFERENCES

[1] AT&T FSM Library[tm] – general-purpose finite-state machine software tools. `http://www.research.att.com/sw/tools/fsm/`.

[2] ABRAMSKY, S. Algorithmic game semantics: A tutorial introduction. Lecture notes, Marktoberdorf International Summer School 2001. (available from `http://web.comlab.ox.ac.uk/oucl/work/samson.abramsky/`), 2001.

[3] ABRAMSKY, S., HONDA, K., AND MCCUSKER, G. A fully abstract game semantics for general references. In *Proceedings, Thirteenth Annual IEEE Symposium on Logic in Computer Science* (1998).

[4] ABRAMSKY, S., JAGADEESAN, R., AND MALACARIA, P. Full abstraction for PCF. *Information and Computation 163* (2000).

[5] ABRAMSKY, S., AND MCCUSKER, G. Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions. vol. 2. 1996, ch. 20, pp. 297–329. Published also as Chapter 20 of [35].

[6] ABRAMSKY, S., AND MCCUSKER, G. Full abstraction for Idealized Algol with passive expressions. *Theoretical Computer Science 227* (1999), 3–42.

[7] ALUR, R., HENZINGER, T. A., AND KUPFERMAN, O. Alternating-time temporal logic. *Journal of the ACM 49*, 5 (Sept. 2002), 672–713.

[8] ALUR, R., HENZINGER, T. A., MANG, F. Y. C., AND QADEER, S. MOCHA: Modularity in model checking. In *Proceedings of CAV'98* (1998), Springer-Verlag, pp. 521–525.

[9] BALL, T., AND RAJAMANI, S. K. The SLAM toolkit. In *13th Conference on Computer Aided Verification (CAV'01)* (July 2001). Available at `http://research.microsoft.com/slam/`.

[10] CLARKE, E. M., GRUMBERG, O., AND PELED, D. A. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[11] COLBY, C., GODEFROID, P., AND JAGADEESAN, L. Automatically closing open reactive programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)* (Montreal, Canada, June 1998), pp. 345–357.

[12] CORBETT, J. C., DWYER, M. B., HATCLIFF, J., LAUBACH, S., PĂSĂREANU, C. S., AND ZHENG, H. Bandera. In *Proceedings of the 22nd International Conference on Software Engineering* (June 2000), ACM Press, pp. 439–448.

[13] DE ALFARO, L., AND HENZINGER, T. A. Interface automata. In *Proceedings of the Joint 8th European Software Engeneering Conference and 9th ACM SIGSOFT Symposium on the Foundation of Software Engeneering (ESEC/FSE-01)*

(New York, Sept. 10–14 2001), V. Gruhn, Ed., vol. 26, 5 of *SOFTWARE ENGINEERING NOTES*, ACM Press, pp. 109–120.

[14] DILL, D. L. The Mur$\phi$ verfication system. In *Proceedings of CAV'96* (1996), vol. 1102 of *LNCS*, Springer-Verlag, pp. 390–393.

[15] GHICA, D. R. A regular-language model for Hoare-style correctness statements. In *Proceedings of the Verification and Computational Logic 2001 Workshop* (Florence, Italy, August 2001).

[16] GHICA, D. R. Regular language semantics for a call-by-value programming language. In *Proceedings of the 17th Annual Conference on Mathematical Foundations of Programming Semantics* (Aarhus, Denmark, May 2001), Electronic Notes in Theoretical Computer Science, Elsevier, pp. 85–98.

[17] GHICA, D. R. *A Games-based Foundation for Compositional Software Model Checking*. PhD thesis, Queen's University School of Computing, Kingston, Ontario, Canada, November 2002. Also available as Oxford University Computing Laboratory Research Report RR-02-13.

[18] GHICA, D. R. Game-based software model checking: Case studies and methodological considerations. Tech. Rep. PRG-RR-03-11, Oxford University Computing Laboratory, May 2003.

[19] GHICA, D. R., AND MCCUKSER, G. The regular-language semantics of first-order Idealized ALGOL. *Theoretical Computer Science* (to appear).

[20] GHICA, D. R., AND MCCUSKER, G. Reasoning about Idealized ALGOL using regular languages. In *Proceedings of 27th International Colloquium on Automata, Languages and Programming ICALP 2000* (2000), vol. 1853 of *LNCS*, Springer-Verlag, pp. 103–116.

[21] HANKIN, C., AND MALACARIA, P. Generalised flowcharts and games. *Lecture Notes in Computer Science 1443* (1998).

[22] HANKIN, C., AND MALACARIA, P. Non-deterministic games and program analysis: an application to security. In *Proceedings, Fourteenth Annual IEEE Symposium on Logic in Computer Science*. 1999, pp. 443–452.

[23] HATCLIFF, J., DWYER, M. B., AND ZHENG, H. Slicing software for model construction. *Higher-Order and Symbolic Computation 13*, 4 (Dec. 2000), 315–353.

[24] HENZINGER, T. A., JHALA, R., MAJUMDAR, R., AND SUTRE, G. Lazy abstraction. In *Proceedings of the 29th Annual Symposium on Principles of Programming Languages* (2002), ACM Press, pp. pp. 58–70.

[25] HOLZMANN, G. J., AND PELED, D. A. The state of SPIN. In *Proceedings of CAV'96* (1996), vol. 1102 of *LNCS*, Springer-Verlag, pp. 385–389.

[26] HOPCROFT, J. E., AND ULLMAN, J. D. *Introduction to Automata Theory, Languages, and Computation*. Addidon Wesley, 1979.

[27] HYLAND, J. M. E., AND ONG, C.-H. L. On full abstraction for PCF: I, II and III. *Information and Computation 163*, 8 (Dec. 2000).

[28] LAIRD, J. Full abstraction for functional languages with control. In *Proceedings, Twelth Annual IEEE Symposium on Logic in Computer Science* (Warsaw, Poland, 29 June–2 July 1997), IEEE Computer Society Press, pp. 58–67.

[29] LAZIC, R., AND NOWAK, D. A unifying approach to data-independence. *Lecture Notes in Computer Science 1877* (2000).

[30] LAZIC, R. S. *A Semantic Study of Data Independence with Applications to Model Checking*. PhD thesis, University of Oxford, 1999.

[31] MANNA, Z., AND PNUELI, A. A hierarchy of temporal properties. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distribted Computing* (Québec City, Québec, Canada, Aug. 1990), C. Dwork, Ed., ACM Press, pp. 377–408.

[32] MURAWSKI, A. S. Complexity of first-order call-by-name program equivalence. submitted for publication, 2003.

[33] MURAWSKI, A. S. On program equivalence in languages with ground-type references. In *Proceedings of LICS'03* (2003), IEEE Computer Society Press. to appear.

[34] MURAWSKI, A. S. Variable scope and call-by-value program equivalence. in preparation, 2003.

[35] O'HEARN, P. W., AND TENNENT, R. D., Eds. ALGOL*-like Languages*. Progress in Theoretical Computer Science. Birkhäuser, Boston, 1997. Two volumes.

[36] ONG, C.-H. L. Observational equivalence of third-order Idealized Algol is decidable. In *Proceedings of IEEE Symposium on Logic in Computer Science, 2002* (July 2002), pp. 245–256.

[37] REAPE, M., AND THOMPSON, H. S. Parallel intersection and serial composition of finite state transducers. *COLING-88* (1988), 535–539.

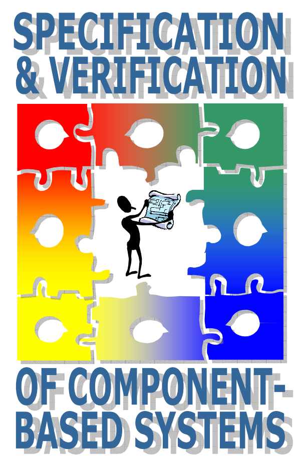
[38] REYNOLDS, J. C. The essence of ALGOL. In *Algorithmic Languages, Proceedings of the International Symposium on Algorithmic Languages* (Amsterdam, Oct. 1981), J. W. de Bakker and J. C. van Vliet, Eds., North-Holland, Amsterdam, pp. 345–372. Reprinted as Chapter 3 of [35].

[39] SCHMIDT, D. A. On the need for a popular formal semantics. *ACM SIGPLAN Notices 32*, 1 (Jan. 1997), 115–116.

[40] SENIZERGUES. L(A) = L(B)? decidability results from complete formal systems. *TCS: Theoretical Computer Science 251* (2001).

[41] STIRLING, C. Deciding DPDA equivalence is primitive recursive. *Lecture Notes in Computer Science 2380* (2002)

# SAVCBS 2003
# POSTER ABSTRACTS



# Bridging the gap between Acme and UML 2.0 for CBD


Miguel Goulão
Departamento de Informática
Faculdade de Ciências e Tecnologia - UNL
2825 Monte de Caparica, Portugal
miguel.goulao@di.fct.unl.pt

Fernando Brito e Abreu
Departamento de Informática
Faculdade de Ciências e Tecnologia - UNL
2825 Monte de Caparica, Portugal
fba@di.fct.unl.pt



## ABSTRACT

Architecture Description Languages (ADLs) such as Acme (a mainstream second generation ADL which contains the most common ADL constructs) provide formality in the description of software architectures, but are not easily reconciled with day-to-day development concerns, thus hampering their adoption by a larger community. UML, on the other hand, has become the de facto standard notation for design modeling, both in industry and in academia. In this paper we map Acme modeling abstractions into UML 2.0, using its new component modeling constructs, its lightweight extension mechanisms and OCL well-formedness rules. The feasibility of this mapping is demonstrated through several examples. This mapping bridges the gap between architectural specification with Acme and UML, namely allowing the transition from architecture to implementation, using UML design models as a middle tier abstraction.



## Keywords

Component-based architectures, component specification, ADLs, Acme, UML.


## 1. INTRODUCTION

Software architectural descriptions provide an abstract representation of the components of software systems and their interactions. There are three main streams of architectural description techniques: ad-hoc, OO techniques and ADLs.

Ad-hoc notations lack formality, preventing architectural descriptions from being analyzed for consistency or completeness and for being traced back and forward to actual implementations [2].

To overcome those drawbacks, one can use ADLs, such as Aesop [3], Adage [4], C2 [5], Darwin [6], Rapide [7], SADL [8], UniCon [9], MetaH [10], or Wright [11]. Although with a considerable overlap on the core, each ADL focuses on different aspects of architectural specification, such as modeling the dynamic behavior of the architecture, or modeling different architectural styles. This diversity provides different approaches to solve specific families of problems. However, the interchange of information between different ADLs becomes a major drawback. Developing a single ADL providing all the features of the various ADLs would be a very complex endeavor. Instead, an ADL called Acme [12] emerged as a generic language which can be used as a common representation of architectural concepts in the interchange of information between specifications with different ADLs [13].

Although ADLs allow for architecture in-depth analysis, their formality is not easily reconciled with day-to-day development concerns. OO approaches to modeling, on the other hand, are more widely accepted in industry. In particular, the UML [14] has become both a *de jure* and *de facto* standard. Using it to describe software architectures could bring economy of scale benefits, better tool support and interoperability, as well as lower training costs.

OO methods have some advantages in the representation of component-based systems, when compared to ADLs. There is a widespread notation, an easier mapping to implementation, better tools support and well-defined development methods. But they also have some shortcomings. For instance, they are less expressive than ADLs when representing connections between components.

Several attempts to map ADLs to UML have been made in the past, as we will see in section 2. One motivation for such attempts is to bring architectural modeling to a larger community, through the use of mainstream modeling notations. Another is to provide automatic refinement mechanisms for architectures. UML can be used as a bridge from architectural to design elements [15]. In this paper we will present a more straightforward mapping from Acme to UML, when compared to previous attempts, due to the usage of the new UML 2.0 metamodel.

We will represent the concepts covered by Acme using the candidate UML 2.0 metamodel, which has been partially approved by the OMG recently. It includes UML's infrastructure [16], superstructure [17] and OCL 2.0 [18]. This increases our modeling power due to the new features of the upcoming standard version, mainly in what concerns the representation of components, ports, interfaces (provided or required), and the hierarchical decomposition of components.

This paper is organized as follows. Related work is discussed in section 2. Section 3 contains a formal specification of the mapping between Acme and UML. Section 4 includes a discussion of the virtues and limitations of that mapping. Section 5 summarizes the conclusions and identifies further work.

## 2. RELATED WORK

A number of mappings among the concepts expressed in ADLs and their representation with UML have been attempted.

A possible strategy is to use UML "as is", in the mapping. In [19], UML is used to express C2 models. In [2], Garlan presents several UML metamodel elements as valid options to express each of the structural constructs defined in Acme. Each mapping becomes the best candidate depending on the goals of the translation from Acme to UML. The semantic mismatch between the ADL and UML concepts is the main drawback of this strategy.

An alternative is to modify the UML metamodel, to increase the semantic accuracy of the mapping [20]. Unfortunately, this drives us away from the standard, and consequently sacrifices existing tool support.

An interesting compromise is to use UML's extension mecha-

nisms to mitigate conceptual mismatches, while maintaining compatibility with the standard metamodel. Examples of this strategy can be found in [15] (C2SADEL to UML), [1] (Acme to UML-RT), and [21] (C2 and Wright to UML). The latter uses OCL constraints on the metamodel elements which is close to the one proposed in this paper, but requires a mapping for each ADL and uses an older and notably less expressive version of UML).

The approach discussed in this paper bridges the gap between software architecture and design using an OO modeling notation. All of the above mentioned mappings were performed with UML 1.x, whereas in our paper we use the new UML 2.0 metamodel elements, which enhance the language's suitability for component-based design.

# 3. MAPING ACME INTO UML

From now on we will assume the reader is familiar with Acme, UML and OCL. Due to space constraints, we omit the OCL definition of predicates such as `IsAcmeComponent()`, `IsAcmeConnector()`, `IsAcmePort()`, `IsAcmeRole()`, `IsAcmeProperty()` and others with self explanatory names that will be used in our mapping presentation. `HasNoOtherInterfaces()` is a predicate that denotes that no other interfaces except for the ones defined in ports will be available for a particular component.

## 3.1 Components

An Acme **component** has **ports**, which act as the component interfaces, **properties**, a **representation** with several bindings (defined as **rep-maps**) and a set of **design rules**. The closest concept in UML is the one of component. To avoid mixing Acme's components with other concepts that we will also represent with UML components, we created a stereotype for Acme components named `<<AcmeComponent>>`, using `Component` as the base class. Invariant 1 assures these components only have interfaces through Acme ports or properties.

```
context Component inv: -- Invariant 1
   self.IsAcmeComponent() implies
      self.ownedPort->forAll(ap|
         ap.IsAcmePort() or
         ap.IsAcmeProperty()) and
      self.HasNoOtherInterfaces()
```

## 3.2 Ports

Acme's ports identify points of interaction between a component and its environment. They can be as simple as operation signatures, or as complex as collections of procedure calls with constraints on the order in which they should be called. UML ports are features of classifiers that specify distinct points of interaction between the classifier (in this case, the component) and its environment (in this case, the rest of the system). UML ports have required and provided interfaces, which can be associated to pre and post conditions. We use a combination of UML port and corresponding required and provided interfaces to express Acme's port concept. Acme ports can only be used with Acme components and they have one provided and one required interface.

```
context Port inv: -- Invariant 2
   self.IsAcmePort() implies
      self.owner.IsAcmeComponent() and
     (self.required->size()=1) and
     (self.provided->size()=1)
```

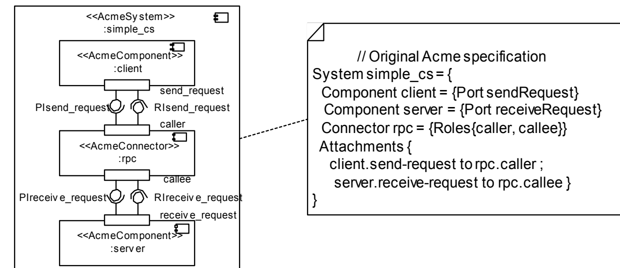
## 3.3 Connectors

Acme **connectors** represent interactions among components. They are viewed as first class elements in the architecture community. Representing them using UML's assembly connector would be visually appealing, but we would loose expressiveness because Acme connectors may be much more complex than a simple interfaces' match. They can be, for example, a protocol, or a SQL link between two components (a client and a database). Moreover, when reusing components built by different teams it is normal that their interfaces do not match exactly. The connector may provide the required glue between the components and this must be made explicit in the design. In order to represent the concept of connector, which has no semantic equivalent in UML, we use a stereotyped component named `<<AcmeConnector>>` and ensure that it has no other interfaces than the ones defined through its roles and properties.

```
context Component inv: -- Invariant 3
   self.IsAcmeConnector() implies
      self.ownedPort->forAll(ap|
         ap.IsAcmeRole() or
         ap.IsAcmeProperty()) and
      self.HasNoOtherInterfaces()
```

Although representing a connector with a stereotyped component clutters the outcoming design, it offers the ability to represent the connector as a first class design element, with flexibility in the definition of any protocols it may implement. Consider the example in Figure 1, where the components `client` and `server` have interfaces that do not match, but the `rpc` connector implements a protocol to make both components interact. We have included provided and required interfaces in both ends of the connector, to illustrate that it provides bi-directional communication abilities.

**Figure 1 – Using the <<AcmeConnector>>**




## 3.4 Roles

In Acme, **roles** are related to connectors the same way as ports are related to components. Thus, it makes sense to represent Acme roles as constrained UML ports, through the use of the `<<AcmeRole>>` stereotype.

```
context Port inv: -- Invariant 4
   self.IsAcmeRole() implies
      self.owner.IsAcmeConnector() and
     (self.required->size()=1) and
     (self.provided->size()=1)
```

## 3.5 Systems

An Acme system represents a graph of interacting components. The UML's concept of package (with the standard `<<subsystem>>` stereotype) represents a set of elements, rather than the structure containing them and is not suitable for defining system-

level properties. To avoid such problems we use the constrained component stereotype `<<AcmeSystem>>`, with the following constraints:

```
context Component inv: -- Invariant 5
   self.IsAcmeSystem() implies
      self.contents()->select(el|
         el.IsKindOf(Component))->asSet()
            ->forAll(comp|
               comp.IsAcmeComponent() or
               comp.IsAcmeConnector())
context Component inv: -- Invariant 6
   self.IsAcmeSystem() implies
      self.contents()->select(el|
         el.IsKindOf(Port))->asSet()
            ->forAll(prt|
               prt.IsAcmePort() or
               prt.IsAcmeRole() or
               prt.IsAcmeProperty())
context Component inv: -- Invariant 7
   self.IsAcmeSystem() implies
      self.ownedPort->forAll(ap|
         ap.IsAcmePort() or
         ap.IsAcmeRole() or
         ap.IsAcmeProperty()) and
      self.HasNoOtherInterfaces()
```

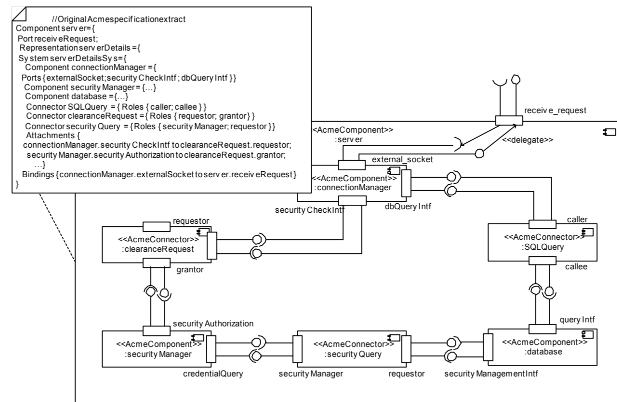
## 3.6 Representations

Acme's **representations** provide the mechanism to add detail to components and connectors. Acme **rep-maps** are used to show how higher and lower-level representations relate to each other. We will use the features for packaging components of UML 2.0 to express representations. UML provides two wiring elements (in the UML specification, they are referred to as "specialized connectors"): `assembly` and `delegation`. The former provides a containment link from the higher level component to its constituent parts, while the latter provides the wiring from higher level provided interfaces to lower level ones, and from lower level required interfaces to higher level ones. A `delegation` corresponds to Acme's rep-map concept. To ensure components are connected to other components through connectors, we need to constrain all assembly connectors to link ports to roles.

```
context connector inv: -- Invariant 8
  self.kind = #assembly implies
    self.end->(exists(cp|cp.role.IsAcmePort())
           and exists(cr|cr.role.IsAcmeRole()) )
```

Figure 2 depicts the specification of `server`. The wiring between the internal structure of server – a system which contains a topology with three components and the connectors among them – and the server's own ports is achieved with the usage of the `<<delegate>>` connectors. Although Acme explicitly uses the concepts of representation and system for defining subsystems, we make them implicit in our mapping. Making them explicit would not improve the expressiveness of the resulting design and would clutter the diagram by creating an extra level of indirection.

**Figure 2 – Detailing a component specification**




## 3.7 Properties

Properties represent semantic information about a system and its architectural elements. To allow automatic reasoning on them, using OCL, we can make these properties available outside the component's internal scope. Ports can be typed with a provided interface that allows the component user to access its properties. The downsides of representing Acme properties as UML ports are that by doing so we are cluttering the design and extending the interfaces provided by the design element. An `<<AcmeProperty>>` port owns a single provided interface that must provide get and set operations for the property's value and type.

```
context Port inv: -- invariant 9
   self.IsAcmeProvided() implies
     (self.required->IsEmpty()) and
     (self.provided->size()=1)
```

## 3.8 Constraints (invariants and heuristics)

Constraints allow the specification of claims on how the architecture and its components are supposed to behave. While invariants are conditions that must hold at all times, heuristics are constraints that should hold, although breaking them is possible. In UML, we can express design constraints through OCL. These constraints can be pre-conditions, post-conditions or invariants. Acme's notion of invariant can be directly mapped to its OCL counterpart. However, there is no direct UML semantic equivalent for the notion of heuristic. This could be circumvented by creating the `<<AcmeConstraint>>` stereotype as a specialization of the UML Constraint metaclass. The former would have an enumerated attribute with two allowed values: `invariant` and `heuristic`.

## 3.9 Styles and Types

An architectural style defines a vocabulary of design elements and the rules for composing them. It is used in the description of families of architectures. Since we have created stereotypes for the several UML constructs used in this Acme to UML mapping, we can now specify architectural styles using these stereotyped elements.

**Figure 3- The pipe and filter family**

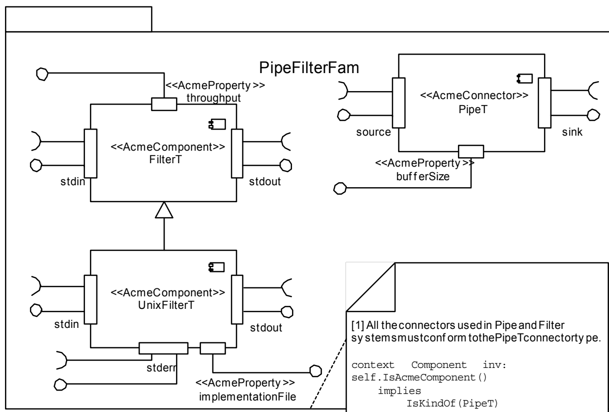



Figure 3 represents the pipe and filter family, an architectural style that defines two types of components, `FilterT` and `UnixFilterT`, a specialization of `FilterT`. The architectural style is defined by means of a UML package, as the family definition does not prescribe a particular topology. It does, however, establish an invariant that states that all the connectors used in a pipe and filter system must conform to `PipeT`.

# 4. DISCUSSION

The presented mapping from Acme to UML is more straightforward than previous approaches. This mainly results from the increased expressiveness provided by the new UML 2.0 design elements. From a structural viewpoint, representing a topology is fairly simple when using UML. This is mainly due to the relative closeness of the sort of structural information that we want to express both at the architectural and design levels. In both cases we have to identify components and the connections among them, possibly at different levels of abstraction.

However, while a connector is regarded as a first class design element by the architecture community, it has no direct mapping in UML 2.0. Our proposal is to promote connectors to first class design elements, by representing them as stereotyped components. This seems to be a good option, considering that the evolution of CBD should provide us with an increasing number of off-the-shelf components and that, the complexity of building component-based software is shifting to the production of glue code. Representing connectors as stereotyped components gives us the extra flexibility to meet this challenge.

The representation of properties is not an easy nut to crack. Perhaps they could be more suitably defined at the meta-level, rather than using the `<<AcmeProperty>>` ports for this purpose, but this still requires further research.

Heuristics are also complex to map directly to UML, as UML provides no direct representation for this concept, although we can use OCL to deal with this problem.

Since Acme does not provide a direct support for component dynamics specification, in this paper we do not address it. Nevertheless, we could use properties to annotate the architectural entities with information on their expected behavior. For instance, a connector may have a property specifying its protocol with some formalism (e.g. Wright). We could use UML's behavioral modeling features similarly, thus complementing the structural information in the mapped specification with a behavioral specification of the design elements used.

# 5. CONCLUSIONS

We have shown the feasibility of expressing architectural information expressed in Acme using the UML 2.0. It is possible to obtain a mapping from a given ADL to UML, through a two-step approach. We could first map the architecture from the original ADL to Acme and then use the mapping proposed in this paper to obtain the corresponding specification in UML. Details lost in the ADL to Acme conversion can always be added later to the resulting UML specification.

The proposed mapping builds upon the added expressiveness of UML 2.0 for architectural concepts, when compared to UML's previous versions. The availability of components with ports typed by provided and required interfaces has proved to be a step forward in the exercise of bridging the gap between architectural and design information. This improves traceability between architectural description and its implementation, using the design as a middle layer between them. This traceability is relevant for keeping the consistency between the architecture, design and implementation of a software system.

The proposed mapping focuses mainly on structural aspects and design constraints. Although it also points out to ways of dealing with the definition of system properties, including semantics and behavioral specification, further research is required to provide more specific guidance on these aspects.

# REFERENCES


[1] S.-W. Cheng and D. Garlan, "Mapping Architectural Concepts to UML-RT", PDPTA'2001, Monte Carlo Resort, Las Vegas, Nevada, USA, 2001.

[2] D. Garlan and A. J. Kompanek, "Reconciling the Needs of Architectural Description with Object-Modeling Notations", <<UML>> 2000, York, UK, 2000.

[3] D. Garlan, R. Allen, and J. Ockerbloom, "Exploiting style in architectural desing environments", SIGSOFT'94: The Second ACM Symposium on the Foundations of Software Engineering, 1994.

[4] L. Coglianese and R. Szymanski, "DSSA-ADAGE: An Environment for Architecture-based Avionics Development", AGARD'93, 1993.

[5] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor, "Using object-oriented typing to support architectural design in the C2 style", SIGSOFT'96: Fourth ACM Symposium on the Foundations of Software Engineering, 1996.

[6] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, "Specifying distributed software architectures", Fifth European Software Engineering Conference, ESEC'95, 1995.

[7] D. C. Luckham, L. M. Augustin, J. J. Kenney, J. Veera, D. Brian, and W. Mann, "Specification and analysis of system architecture using Rapide", *IEEE Transactions on Software Engineering*, vol. 21, No.4, pp. 336-355, 1995.

[8] M. Moriconi, X. Qian, and R. Riemenschneider, "Correct architecture refinement", *IEEE Transactions on Software Engineering*, vol. 21, No. 4, pp. 356-373, 1995.

[9] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik, "Abstractions for Software Architecture and Tools to support them", *IEEE Transactions on Software Engineering*, vol. 21, No. 4, pp. 314-335, 1995.

[10] P. Binns and S. Vestal, "Formal real-time architecture specification and analysis", Tenth IEEE Workshop on Real-Time Operating Systems and Software, New York, USA, 1993.

[11] R. Allen and D. Garlan, "A Formal Basis for Architectural Connection", *ACM Transactions on Software Engineering and Methodology*, vol. 6, pp. 213-249, 1997.

[12] D. Garlan, R. T. Monroe, and D. Wile, "Acme: Architectural Description of Component-Based Systems", in *Foundations of Component Based Systems*, G. T. Leavens and M. Sitaraman, Eds.: Cambridge University Press, 2000, pp. 47-68.

[13] M. R. Barbacci and C. B. Weinstock, "Mapping MetaH into ACME", Carneggie Mellon University / Software Engineering Institute, Technical Report CMU/SEI-98-SR-006, July 1998.

[14] OMG, "OMG Unified Modeling Language Specification. Version 1.5", Object Management Group March 2003.

[15] A. Egyed and N. Medvidovic, "Consistent Architectural Refinement and Evolution using the Unified Modeling Language", 1st Workshop on Describing Software Architecture with UML, co-located with ICSE 2001, Toronto, Canada, 2001.

[16] U2-Partners, "3rd revised submission to OMG RFP ad/00-09-01: Unified Modeling Language: Infrastructure - version 2.0", U2-Partners January 2003.

[17] U2-Partners, "2nd revised submission to OMG RFP ad/00-09-02: Unified Modeling Language: Superstructure - version 2.0", U2-Partners January 2003.

[18] Boldsoft, Rational, IONA, and Adaptive, "Response to the UML 2.0 OCL RfP (ad/2000-09-03) - Revised Submission, Version 1.6 - OMG Document ad/2003-01-07", OMG 2003.

[19] N. Medvidovic and D. S. Rosenblum, "Assessing the Suitability of a Standard Design Method for Modeling Software Architectures", First Working IFIP Conference on Software Architecture, 1999.

[20] B. Selic, "On Modeling Architectural Structures with UML", ICSE 2002 Workshop Methods and Techniques for Software Architecture Review and Assessment, Orlando, Florida, USA, 2002.

[21] J. E. Robbins, N. Medvidovic, D. F. Redmiles, and D. S. Rosenblum, "Integrating Architecture Description Languages with a Standard Design Method", International Conference on Software Engineering (ICSE98), Kyoto, Japan, 1998.

# Abstract OO Big O

Joan Krone
Denison University
Department of Math and CS
Granville, Ohio 43023
740-587-6484
krone@denison.edu

W. F. Ogden
The Ohio State University
Neal Avenue
Columbus, Ohio 43210
614-292-6007
ogden@cis.ohio-state.edu

## SUMMARY

When traditional Big O analysis is rigorously applied to object oriented software, several deficiencies quickly manifest themselves. Because the traditional definition of Big O is expressed in terms of natural numbers, rich mathematical models of objects must be projected down to the natural numbers, which entails a significant loss of precision beyond that intrinsic to order of magnitude estimation. Moreover, given that larger objects are composed of smaller objects, the lack of a general method of formulating an appropriate natural number projection for a larger object from the projections for its constituent objects constitutes a barrier to compositional performance analysis.

We recast the definition of Big O in a form that is directly applicable to whatever mathematical model may have been used to describe the functional capabilities of a class of objects. This generalized definition retains the useful properties of the natural number based definition but offers increased precision as well as compositional properties appropriate for object based components. Because both share a common mathematical model, functional and durational specifications can now be included in the code for object operations and formally verified. With this approach, Big O specifications for software graduate from the status of hand waving claim to that of rigorous software characterization.

## Categories and Subject Descriptors

D.2[Software Engineering], F.2[Analysis of Algorithms], F.3[Logics and Meanings of Programs]: Specifications, Models, Semantics – *performance specifications, performance analysis, performance proof rules.*

## General Terms

Algorithms, Performance, Verification.

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, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*Conference '00*, Month 1-2, 2000, City, State.
Copyright 2000 ACM 1-58113-000-0/00/0000…$5.00.

## Keywords

Performance, Formal Specification, Verification, Big O.

## 1. INTRODUCTION

The past forty years have seen a great deal of work on the rigorous specification and verification of programs' functional correctness properties [2] but relatively little on their performance characteristics. Currently performance "specifications" for programs commonly consist of reports on a few sample timings and a general order of magnitude claim formulated in a Big O notation borrowed from number theory. As we have discussed elsewhere [6], such an approach to the performance of reusable components is no more adequate than the test and patch approach is to their functionality.

As with functionality, problems with performance usually have their roots in the design phase of software development, and it's there that order of magnitude considerations are most appropriately encountered. This means our order of magnitude notations are generally applied in a somewhat rough and ready fashion (which is probably why the deficiencies of our current ones have escaped notice for so long). However, if their formulation doesn't reflect the ultimate performance of the components under design accurately and comprehensibly, then marginal designs become almost inevitable. So the way to get an appropriate order of magnitude definition is to formulate one that meshes smoothly with program verification.

With the advent of object oriented programming and a component based approach to software, formal specifications of a component's functionality are considered to be critical in order for clients to make good choices when putting together a piece of software from certified components.

To meet the need for reasoning about performance as well as functionality, we introduce a new object appropriate definition of Big O. Object Oriented Big O, or OO Big O for short, allows one to make sensitive comparisons of running times defined over complex object domains, thereby achieving much more realistic bounds than are possible with traditional big O.

We cast our approach in a framework that includes an assertive language with syntactic slots for specifying both functionality and performance, along with automatable proof rules that deal with both. Equally important is the need for the reasoning to be fully modular, i.e., once a component has been certified as

correct, it should not be necessary to reverify it when it is selected for use in a particular system.

Our approach is based on the software engineering philosophy that a component should be designed for reuse and thus include a general mathematical specification that admits several possible implementations – each with different performance characteristics [3, 7]. Of course, in order for a component to be reusable, it should include precise descriptions of both its functionality and its performance, so that prospective clients can be certain they are choosing components that fit their needs.

It is also important that all reasoning about constituent components – including reasoning about performance – be possible without knowing implementation details. In fact, if one is using generic components, it should be possible to reason about those components even before details such as entry types to a generic container type, are available.

## 2. OO Big O Definition

The traditional Big O is a relation between natural number based functions defined in the following way:
Given f, g: $\mathbb{N} \to \mathbb{R}$, f(n) is O(g(n)) iff ∃ constants c and $n_0$ such that $f(n) \leq c \cdot g(n)$ whenever $n \geq n_0$. A program whose running time is O(g) is said to have growth rate g(n) [1].

When the object classes central to modern programming are formally modeled, they present to clients a view of objects that could come from essentially arbitrary mathematical domains, since the point of introducing objects is to simplify reasoning about the functionality of components by providing a minimalist explanation that hides the details of their implementation. But the functionally simplest models may well have little to do with natural numbers. So the natural expression of the duration f(x) of an operation on object x is as a function directly from the input domain used to model x to the real numbers. Clearly any gauging function g that we might want to use as an estimate for f should have this same domain. Accordingly, the Is_O relation between functions ( f(x) Is_O g(x) ) is defined by:

**Definition**: $(f: Dom \to \mathbb{R})$ Is_O $(g: Dom \to \mathbb{R})$: $\mathbb{B}$ = ( $\exists$ A: $\mathbb{R}^{>0}$, $\exists$ H: $\mathbb{R}$ ∍ $\forall$ x: Dom, $f(x) \leq A \cdot g(x) + H$ ).

In other words, for two timing functions f and g mapping a computational domain Dom to the real numbers, to say that f(x) Is_O g(x) is to say that there is some positive acceleration factor A and some handicap H such that for every domain value x, $f(x) \leq A \cdot g(x) + H$. If we think of f and g as representing competing processes, f being big O of g means that g is not essentially faster than f. If f is run on a processor A times faster than g's processor and also given a head start H, then f will beat g on all input data x.

Of course, in order to use this definition, it is necessary to have mathematical support in the form of theorems about the revised definition of Is_O. For example, we need an additive property so we can apply our analysis to a succession of operation invocations:

**Theorem** OM1: If $f_1(x)$ Is_O $g_1(x)$ and $f_2(x)$ Is_O $g_2(x)$, then $f_1(x) + f_2(x)$ Is_O $Max(g_1(x), g_2(x))$.

A development of appropriate theorems and definitions appears in [5].

## 3. ABSTRACT OBJECTS

If you want to produce rigorously specified and verified software components that support genericity, facilitate information hiding, and can be reasoned about in a modular fashion, it is necessary to adhere carefully to certain guidelines and principles [7].

A simple example of a general purpose component concept that can be used to make clear the need for a new definition of Big_O is the one that captures the "linked list." Because one of our guidelines is to tailor a concept to simplify the client's view, we call this concept a one-way list template and the objects it provides list positions. We describe list positions mathematically as pairs of strings over the entry type. The first string in a list position contains the list entries preceding the current position and is named *Prec*; the second string is the remainder of the list, *Rem*. Since the operations on list position (Insert, Advance, Reset, Remove, etc.) all have easy specifications in terms of this model, and since the underlying linking pointers are cleanly hidden, reasoning about client code is much simplified with this abstract model.

```
Concept One_Way_List_Template( type Entry;
                    evaluates Max_Total_Length: Integer );
        uses Std_Integer_Fac, String_Theory;
        requires Max_Total_Length > 0;
  Type Family List_Position ⊆ Cart-Prod
                              Prec, Rem: Str(Entry)
                          end;
        M
  Operation Advance( updates P: List_Position );
    requires P.Rem ≠ Λ;
    ensures P.Prec∘P.Rem = @P.Prec∘@P.Rem and
                              |P.Prec| = |@P.Prec| + 1;
```

Although variations in list implementation details are usually insignificant, our system allows for the possibility of a multiplicity of different realizations (implementations) for any given concept. Each **Realization**, with its own potentially distinct performance characteristics, retains a generic character, since parameter values such as the entry type for lists have yet to be specified. The binding of such parameters only takes place when a client makes a **Facility**, which involves selecting the concept and one of its realizations along with identifying the appropriate parameters.

When designing concepts for maximal reusability, our guidelines prescribe that only the basic operations on a class of objects should be included, so for lists we only include Insert, Advance, etc., but not Search, Sort, etc. In order to have a rich enough Big O example, we will consider such a sorting operation, so we need to employ the **Enhancement** construct used to enrich basic concepts such as the one-way list.

Well-designed enhancements also retain to the extent possible the generality we seek in our concepts, but often they do add constraints that prevent their use in certain situations. Providing a Sort_List operation, for example, requires that list entries possess an ordering relation $\precsim$, so certain classes of entries would be precluded from lists if Sort_List were one an operation in the basic list concept.

## EXAMPLE APPLICATION OF BIG O

The enhancement's name here is *Sort_Capability*, and it maintains the generic character of the concept (which allows entries to be of arbitrary type) by importing an ordering relation ≾ on whatever the entry type may be. A **requires** clause insists that any imported ≾ relation actually be a total preordering on whatever the entry type is.

The **uses** clause indicates that this component relies on a mathematical theory of order relations for the definitions and properties of notions such as total preordering. Note that an automated verifier needs such information.

**Enhancement** Sort_Capability( **def**. **const** (x: Entry) ≾
(y: Entry): B
);
**for** One_Way_List_Template;
**uses** Basic_Ordering_theory;
**requires** Is_Total_Preordering( ≾ );
**Def**. **const** In_Ascending_Order( α: Str(Entry) ): B =
( ∀ x, y: Entry, **if** ⟨x⟩∘⟨y⟩ Is_Substring α, **then** x ≾ y );
**Operation** Sort_List( **updates** P: List_Position );
**ensures** P.Prec = Λ **and** In_Ascending_Order( P.Rem ) **and**
P.Rem Is_Permutation @P.Prec∘@P.Rem;
**end** Sort_Capability;

A client who wishes to order a list would be able to choose this list enhancement on the basis of these functional specifications. However, before choosing among the numerous realizations for it, a client should be able to see information about their performance. Rather than giving such timing (**duration**) information a separate ad hoc treatment, we introduce syntax for formally specifying **duration** as part of each **realization.** In short, we associate with every component not only a formal specification of its functionality but of its performance as well, so that a potential client can choose a component based on its formal specifications rather than on its detailed code.

To see how the new Big O definition can improve performance specifications, we consider an insertion sort realization for the Sort_List operation.

Because a realization for a concept enhancement relies upon the basic operations provided by the concept, its performance is clearly dependent on their performance, and that can vary with the realization chosen for the concept. Fortunately performance variations for a given concept's realizations seem to cluster into parameterizable categories, which we can capture in the **Duration Situation** syntactic slot. The normal situation for a one-way list realization, for example, is that the duration of each operation Is_O(1). Of course realizations of lists with much worse performance are possible, but we wouldn't ordinarily bother to set up a separate duration situation to support analyzing their impact on our sort realization.

Duration situations talk about the durations of supporting operations such as the Insert and Advance operations by using the notation $\mathbf{Dur}_{\text{Insert}}$(E, P), $\mathbf{Dur}_{\text{Advance}}$(P), etc. So we can use our Big O notation to indicate that the performance estimates labeled "normal" only hold when $\mathbf{Dur}_{\text{Insert}}$(E, P) **Is_O** 1, etc.

**Realization** Insertion_Sort_Realiz(
**Oper** Lss_or_Comp( **restores** E1, E2: Entry ): Boolean;
**ensures** Lss_or_Comp = ( E1 ≾ E2 ); )
**for** Sorting_Capability;
**Duration Situation** Normal: $\mathbf{Dur}_{\text{Insert}}$(P) **Is_O** 1 **and**
$\mathbf{Dur}_{\text{Advance}}$(P) **Is_O** 1 **and** $\mathbf{Dur}_{=}$(i, j) **Is_O** 1 **and** Λ
**Inductive def**. **on** α: Str(Entry) **of const**
Rank( E: Entry, α ): ℕ **is**
(i) Rank( E, Λ ) = 0;

$$\text{(ii) Rank( E, ext}(\alpha, D)\text{ ) } = \begin{cases} \text{Rank}(E,\alpha)+1 & \text{if } D\ \pi\ E \\ \text{Rank}(E,\alpha) & \text{otherwise} \end{cases};$$

M
**Inductive def**. **on** α: Str(Entry) **of const** P_Rank( α ): ℕ **is**
(i ) P_Rank( Λ ) = 0;
(ii) P_Rank( ext(α, E) ) = P_Rank( α ) + Rank( E, α );
**Theorem** IS6: ∀ β: Str(Entry), P_Rank( β ) ≤ |β|·(|β| − 1)/2;
**Def**. **const** Len( P: List_Position ): ℕ = ( |P.Prec∘P.Rem| );
**Proc** Sort_List( **updates** P: List_Position );
**Duration** Normal:
**Is_O** Max( Len( @P ), P_Rank(@P.Prec∘@P.Rem) );
**Var** P_Entry, S_Entry: Entry;
M
**While** Length_of_Rem( P ) ≠ 0
**affecting** P, P_Entry, Sorted, S_Entry, Processed_P;
**maintaining** Sorted.Prec = Λ **and**
In_Ascending_Order( Sorted.Rem ) **and**
Processed_P.Prec∘P.Rem = @P.Prec∘@P.Rem **and**
Sorted.Rem Is_Permutation Processed_P.Prec;
**decreasing** |P.Rem|;
**elapsed_time** Normal: **Is_O** P_Rank(Processed_P.Prec)
+ |Processed_P.Prec|;
**do**
Remove( P_Entry, P );
M

For each loop, we record the loop invariant that is used to establish the functional effect in its the **maintaining** clause, while the progress metric used to prove termination is recorded in the **decreasing** clause. The **elapsed_time** clause is used to specify on each pass through the loop how much time has elapsed since the loop was entered, which can vary according to the named situation ("Normal" in our example). If an elapsed time clause begins with the **Is_O** token, then the verifier must establish that the actual elapsed time function Is_O of the gauge function specified by the subsequent expression.

The portion of our proof rules that deals with verifying duration estimates for loops accomplishes its objective by checking that the duration of each loop body Is_O of the difference between the value of the gauge function at the end of the loop body and its value at the beginning.

Clearly the elapsed time of an insertion sort depends heavily upon the order of the elements in the original list @P, but traditional natural number based Big_O analysis would require that we project the @P list onto a natural number "n" and express our gauge function in terms of that n (e.g. $n^3$). Typically that n would be the length of a list (what we've formally defined as Len(P) so that n = Len(@P)). Since Len(@P) is totally insensitive to the order of the entries in @P, we could at best end up with a duration estimate for Sort_List of $n^2$.

To exploit the increased precision of the OO Big O definition, we need to define a function on strings of entries α that counts

how many entries in $\alpha$ are less than an entry E and hence would be skipped over when positioning E after $\alpha$ has been sorted, and that's why our realization includes the definition of the Rank( E, $\alpha$ ) function. Since the elapsed time of the outer loop depends upon the cumulative effect of positioning successive entries in @P, we also need to define a "preceding rank" function P_Rank( $\alpha$ ).

Using these definitions, we can express **elapsed time** bounds for the two loops in the code and the overall **Duration** bound:

Max( Len( @P ), P_Rank(@P.Prec∘@P.Rem) ).

Now one of the theorems about P_Rank is that $\text{P_Rank}(\alpha) \leq |\alpha|\cdot(|\alpha|-1)/2$, so it follows that $\mathbf{Dur}_{\text{Sort_List}}(\text{P})$ Is_O $\text{Len(P)}^2$ too, and we can get the much less exacting estimate produced by traditional Big O analysis if we wish. We're just not forced to when we need a sharper estimate. Another point to note is that besides being compatible with correctness proofs for components, the direct style of performance specification is much more natural than the old style using the often ill defined "n" as an intermediary.

## 4. THE CALCULUS FOR OO BIG O

Our Sort_List example illustrates how we can use the new Big O notation in performance specifications and indicates how such specifications could fit into a formal program verification system. The success of such a verification system depends upon having a high level calculus for Big O that allows verification of performance correctness to proceed without direct reference to the detailed definition of Big O.

Of course making such a calculus possible is one of the primary motivations for the new Big O definition, and in [4] we have developed a number of results like the earlier theorem OM1 to support this calculus. Another simple illustration of a property of the new Big O important for verification is dimensional insensitivity.

Theorem OM2: If f(x) Is_O g(x) and F(x, y) = f(x) and

G(x, y) = g(x), then F(x, y) Is_O G(x, y).

Taken together, these results must justify both the proof rules for our program verification system and the expression simplification rules for the resulting verification conditions.

## 5. CONCLUSION

A critical aspect of reusable components is assured correctness, an attribute attainable only with formal specifications and an accompanying proof system. Here, we claim that while functional correctness is absolutely necessary for any component that is to be reused, it is not sufficient. Reusable components need formally specified performance characteristics as well.

Traditional Big O order of magnitude estimates are inadequate because they deal only with the domain of natural numbers and offer no support for modularity and scalability.

If we want to design software components that can be reused, such components must have formal specifications for both functionality and performance associated with them and there must be a proof system that addresses both. Moreover, to avoid intractable complexity, it must be possible to reason about these components in a modular fashion, so that one can put together hierarchically structured programs, each part of which can be reasoned about using only the abstract specifications for its constituent parts. To avoid the rapid compounding of imprecision that otherwise happens in such systems, it is also essential to use high precision performance specification mechanisms such as OO Big O.

To develop maximally reusable components, it is necessary to be able to reason about them in a generic form, without knowing what parametric values may be filled in when the component is put into use.

OO Big O satisfies all these criteria, supporting complete genericity, performance analysis of programs over any domain, and modular reasoning.

## 6. REFERENCES


1. Aho, A., Hopcroft, J., Ullman, J., Data Structures and Algorithms, Addison-Wesley, 1983.
2. de Roever, W., Engelhardt, K. Data Refinement: Model-Oriented Proof Methods and their Comparison, Cambridge University Press, 1998.
3. Krone, "The Role of Verification in Software Reusability." Dissertation, The Ohio State University, 1988.
4. J. Krone, W. F. Ogden, and, M. Sitaraman, Modular Verification of Performance Constraints, Technical Report RSRG-03-04, Department of Computer Science, Clemson University, Clemson, SC 29634-0974, May 2003, 25 pages.
5. Ogden, W. F., CIS680 Coursenotes, Spring 2002.
6. Sitaraman, M., Krone, J., Kulczycki, G., Ogden, W., and Reddy, A. L. N., "Performance Specification of Software Components," *ACM SIGSOFT Symposium on Software Reuse*, May 2001.
7. Weide, B., Ogden, W., Zweben, S., "Reusable Software Components," in M.C. Yovits, editor, **Advances in Computers**, Vol 33, Academic Press, 1991, pp. 1 – 65

# Ontology-based Description and Reasoning for Component-based Development on the Web

Claus Pahl
Dublin City University, School of Computing
Dublin 9, Ireland

## ABSTRACT

Substantial efforts are currently been made to transform the Web from a document-oriented platform for human consumption into a software-oriented application-to-application platform. The Web Services Framework provides the necessary languages, protocols, and support techniques. Even though Web services exhibit a basic component model, much can be gained by fully embracing software component technology for the Web. We propose to base this endeavour on ontology technology – an integral element of the Semantic Web. We will introduce an ontology that in particular provides a rich reasoning framework for behavioural aspects of Web services or, indeed, components on the Web.

## 1. THE WEB AND SOFTWARE DEVELOPMENT AND DEPLOYMENT

The Web is undergoing dramatic changes at the moment. From a human-oriented document publishing framework it has more and more developed into a platform where we can equally well find software applications. The application-to-application use of the Web is one of the recent endeavours to enhance the capabilities of the platform. The *Web Services* initiative [18] bundles these efforts to provide software applications in form of targeted services.

The current Web is a platform comprising *description languages* (such as HTML), *protocols* (such as HTTP), and *tools* (such as browsers and search engines) to support search, retrieval, transportation, and display of documents. The Web Services Framework provides a similar set of technologies – a description language WSDL (Web Service Description Language) for software services, a protocol SOAP (Simple Object Access Protocol) for service interactions, and tool support in form of UDDI (Universal Description, Discovery, and Integration Service) – a registry and marketplace where providers and users of Web services can meet.

Web services are important for middleware architectures. Various architectures, e.g. CORBA, have been established, but interoperability between these individual architectures, in particular in distributed environments, is still a major problem. Web services can, due to the ubiquity of the Web, provide interoperability.

Clearly, Web Services can encapsulate software components from various architectures and provide uniform Web-based interfaces and Web-based communication infrastructures for the component deployment. Web services themselves exhibit a simple *component model* [5, 16]. Even though service deployment has been the focus so far, the support of component-style deployment is, however, only one aspect of the Web Services Framework. We would like to emphasise here the importance of the development aspect – *component-based software development* using the *Web as the development platform* is often neglected or treated as a secondary aspect. Component development for the Web and using the Web requires the support by specific Web technologies that we will discuss here.

Besides services at the core of the Web Service Framework, we also look at current trends in this area. In particular Web service coordination creating service processes and interactions is an aspect that has received much attention [10, 1], and that is also of importance from a component-oriented perspective.

## 2. COMPONENT-BASED SOFTWARE DEVELOPMENT FOR THE WEB

Component-based software development [9, 16] is an ideal technology to support Web-based software development. As already mentioned, Web Services are based on a simple component model. A service encapsulates a coherent collection of operations. A central objective of component technology – central also for Web services – is the separation of computation and connection. *Computational aspects* are abstracted by interface descriptions. Usually, a set of semantically described, coherent operations forms an interface. *Connection* is more than plugging a provided operation into a requested operation – the coordination (or choreography) of services and their operations to more complex service processes is another important aspect of connectivity.

The (automated) interaction between component providers and clients becomes crucial in this context. Due to the nature of the Web platform, automation of these processes is highly important. The description of provided and required services needs to be supported. Reasoning to support matching of provided and required services is essential. Two aspects shall be distinguished:

- *Component connection and interaction* is based on plugging components together when a client requests services of a provider, possibly involving connectors or glue code. The semantical description of both provided and required component (or service) interfaces is essential in the Web context.

- The composition of services to *coordinated service processes* exhibiting a predefined *interaction pattern* is the other aspect [13]. Several extensions of the Web Services framework in this direction have already been made – such as the Web Ser-

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.
*SAVCBS'03 - ESEC/FSE'03 Workshop,* Sept 1-2, 2003, Helsinki, Finland.
Copyright 2003 ACM 1-58113-743-5/03/0009 ...$5.00.

vices Flow Language WSFL [10] or the Web Services Coordination Language WSCL [1].

The architecture of this approach is illustrated in Figure 1. Critical for both forms is development support for a style of software development that is strongly based on retrieving and assembling suitable off-the-shelf components from a range of different component repositories. Reuse is the central theme for component development on the Web. The heterogeneity, distribution, and decentralisation requires a high degree of robustness and dependability of software components for the Web.

# 3. ONTOLOGIES AND COMPONENTS

## 3.1 Semantics and Knowledge

Describing the semantics of components means to express knowledge about the behaviour or other, non-functional aspects of a component or component process. This knowledge comprises:

- *Domain knowledge* from the application domain – basic entities and their properties are defined. This aspect – usually known as domain modelling in requirements engineering – is a widely accepted method.
- *Software-related knowledge* in relation to the type of semantics – behavioural semantics could express the behaviour of operations in a state-based system, techniques such as refinement could be defined. This is – due to the distributed and collaborative development on the Web – an emerging aspect.

## 3.2 The Semantic Web

In the context of *component-based Web service development*, the description of knowledge and also reasoning about knowledge are essential requirements. This requirement, in principle, can be satisfied through the techniques of another activity in the Web context – the Semantic Web initiative [18]. Description of and reasoning about knowledge is the objective of *ontology technology*, which is at the core of the Semantic Web. It provides XML-based knowledge description and ontology languages. Ontology languages facilitate the hierarchical description of concepts from a domain and their properties and support reasoning about these concepts and properties through a suitable logic.

The aim of the *Semantic Web* is to open the Web to processing by software agents. The exploitation of the full potential of the Web as an information network is currently hampered by the fact that information is provided for human consumption. Software processing, e.g. searches using search engines, is often inaccurate and error-prone. Adding semantical descriptions to Web resources and logic to reason about properties based on *ontologies shared between Web users* is the key contribution of the Semantic Web.

The application of ontologies is certainly not limited to application domains; they can also be used to capture software development principles and reasoning techniques.

## 3.3 Ontology Languages

The Semantic Web is based on the Resource Description Framework RDF – an XML-based language to express properties in terms of triples (Subject, Property, Object) [18]. Subjects (or concepts) are defined in terms of their properties in relation to other, already defined objects (concepts). We could, for instance, say that a component has an author, `(Component, hasAuthor, Author)`. In this way, based on some basic concepts, a hierarchy of complex, semantically defined concepts can be created. The ontology language DAML+OIL [18] (most likely the future Ontology Web Language OWL) is an extension of RDF by a rich set of operators and features to support ontology description and reasoning.

Reasoning is a central aspect that needs to be supported by a suitable logic. DAML+OIL is essentially a very expressive description logic. Description logics [2] are first-order logics that provide operators to specify concepts and their properties.

## 3.4 Ontology Support for Component-based Service Description and Composition

### 3.4.1 Description – Interface and Interaction

Ontology languages usually support the notions of concepts and properties (or roles)[1]. Ontology languages are application-domain independent. In the context of component-based Web services, the first essential question is what the description logic concepts and roles represent. An intuitive choice might be to use concepts to represent services or operations, and to express their properties using roles. We, however, suggest a different approach (Fig. 2). *Concepts* represent *descriptions of service properties*. *Roles* are used to represent the *services* themselves. Roles are usually interpreted as relations on classes of individuals (that represent concepts) – here they are interpreted as accessibility relations on states. This choice enables us to view a description logic specification as the specification of a state-based system with descriptions of state properties through concepts and specification of state transitions through roles. We actually distinguish two role types. *Descriptional roles* correspond to the classical use of roles as properties – examples in our ontology are preCond, postCond, opName, or opDescr, see Fig. 2. *Transitional roles* are roles representing operations, as we have just introduced.

Roles – supposed to represent services and operations here – are classically used in description logics to express simple concept properties. For instance, for a concept `Component`, the term $\forall$`hasAuthor.Author` is a *value restriction* that says that all components have authors. For a concept `State`, the *existentially quantified expression* $\exists$`preCond.valid(input)` says that for a given class of states, there is at least one for with a precondition `valid(input)` holds. Concepts are usually interpreted by classes of objects (called individuals). Both `hasAuthor` and `preCond` are roles – interpreted by relations on classes of individuals. $\forall$`update.`$\forall$`postCond.equal(retrieve(id),doc)` means that by executing operation `update` a state is reached that is described by postcondition `equal(retrieve(id),doc)`.

Even though some extensions of description logics exist in the literature [2], special attention has to be dedicated to roles if the needs arising from the application of ontology technology to components and services as suggested here have to be addressed. Elements of the language that need attention are: operations and parameters, service processes, and interactions of (provided and requested) services. We have developed a description logic that satisfies these criteria [14] – see Figure 2. At the core of this logic is a rich *role expression sublanguage*.

*Operations – names and parameters*: Usually, individuals can be named in description logics, but the notion of a variable or an abstract name is unknown. We have introduced names as roles, since they are here required as part of role expressions, interpreted as constant functions. Parameterisation is expressed through functional (sequential) composition of roles. We can express a parameterised role such as $\forall$`Login` $\circ$ `id.post` where `id` is a name that is a parameter to operation (role) `Login`.

[1]We focus here on description logic as the underlying ontology language – instead of the more verbose DAML+OIL.

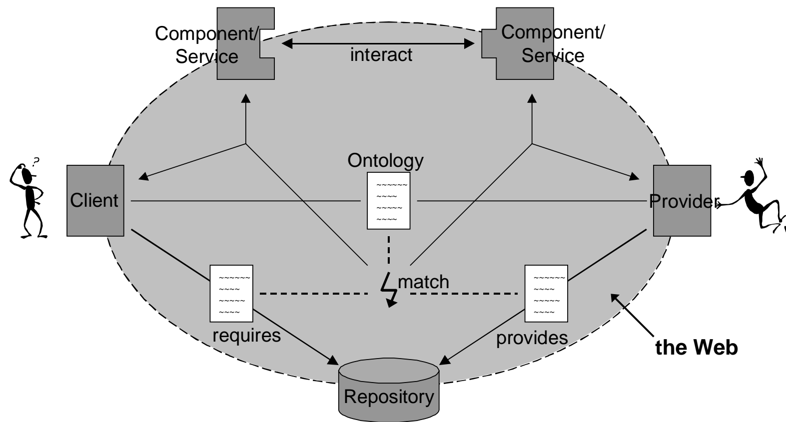



**Figure 1: Ontology-based Component Development for the Web**

*Processes – role expressions*: Besides the sequential (functional) composition of roles (representing operations and parameters), other forms of composition of services to service processes are required [14]. These composition operators include non-deterministic choice, iteration, and parallel composition; the following is an example `∀Login; !(Catalog + Quote); Purchase.post`. The semantics of these operators can be defined in a transition system-based semantics for the logic.

*Interaction – ports and the service life cycle*: It is important to cover all aspects of the composition process including the matching of specifications, the connection of services, and the actual interaction between services [13, 14]. This is actually a central part of the life cycle of a service. A more network-oriented notion of ports, representing services in a component architecture is the central notion. Send and receive operations need to be distinguished – which means for the logic that role names can appear in these two forms.

The central aim of bringing ontology technology to component and Web service development is in enable meaningful, knowledge-level interaction through semantically decribed interfaces. Ontologies provide domain- and software development-related knowledge representation techniques for interfaces. Knowledge describing interfaces enables meaningful interaction.

### *3.4.2 Reasoning – Support for Service Matching*

Reasoning in description logic is based on the central idea of *subsumption* – which is the subclass (or subset) relationships between classes that interpret either concepts (sets) or roles (relations). A composition activity that requires reasoning support in this context is *service or component matching*. The compatibility of a service requirements specification and the description of a provided service has to be verified.

Three research aspects are important in relation to matching support. Firstly, the subsumption idea has to be related to suitable *matching notions* for components and services. Secondly, the specific nature of *roles* and *role expressions* representing (composite) operations has to be addressed. Thirdly, the *tractability* of the result logical framework has to be considered. These research issues shall now be looked at in more detail.

*Subsumption and matching*: Matching between specifications of required and provided services can be expressed in form of classical notion used in computing, such as refinement or simulation [14]. For service interfaces we propose a *refinement* notion, which, if based on a design-by-contract matching notion on pre- and post-conditions, can be proven to imply subsumption [3]. For service process we propose a *simulation* notion on role expressions, which can also be proven to imply subsumption.

*Role expressions and transitions*: Roles are used to represent operations, i.e. roles have a transitional character. Essential is here suitable reasoning support for transitional roles. A *link* between *description logic* and *dynamic logic* (a logic of programs [7]) provides this support. Schild [15] investigates a correspondence between role-based concept descriptions in description logic and modal operators in dynamic logic. This correspondence allows us to adapt modal logic reasoning about state-based system behaviour into description logics. This correspondence is the essential benefit of chosing to represent services as roles, and not as concepts. Aspects of process calculi such as simulation notions can also be adapted for the description logic context [14].

*Tractability*: Tool support and automation are critical points for the success of the proposed component-based Web service development and deployment framework. Therefore, the tractability and a high degree of automation are desirable – even tough difficult to achieve. *Decidability and complexity* tend cause tractability problems in various logics. Here, we have proposed a *very expressive description logic*. Some properties of the proposed framework, however, seem to indicate that these problems can be adequately addressed. Transitional roles can be limited to interpretation by functions. Negation as an operator (known to cause difficulty) is not required for role expressions. Furthermore, application domain ontologies can be integrated through admissible concrete domains.

The specification of service and operation semantics often involves concepts from the application domain. Description logic theory [2] introduces a technique called *concrete domains* to handle the intoduction of specific classes of objects and their predicates into an existing semantics – an abstract example would be a number domain with its predicates. In order to retain the decidability in this framework, an *admissibility* criterion has to be satisfied. We have shown that this is possible for standard application domains such as numerical domains and domains of similar complexity [14].

## 4. RELATED WORK

*DAML-S* [6] is an ontology for Web services. DAML-S supports a range of descriptions for different service aspects from textual descriptions to behavioural semantics. The central difference between DAML-S and our framework it that DAML-S models services as

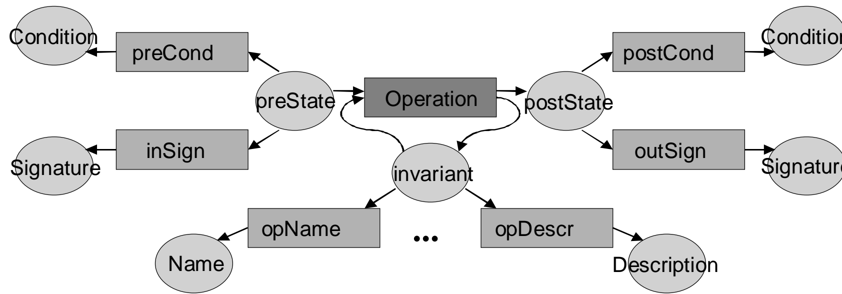



**Figure 2: Service Process Ontology focussing on Operations**

concepts, whereas we model services (more precisely operations of a service) as roles. The essential benefit of our approach is the correspondence between description logics and dynamic logic. This correspondence allows us to establish a richer reasoning framework in particular for the behavioural aspects of service descriptions.

The correspondence between *description logic and dynamic logic* was explored by Schild [15] around a decade ago, but has, despite its potential, not found its way into recent applications of ontology technology for software development. We have enhanced a description logic by results from two other software-engineering related areas – *modal logics* [7] and *process calculi*. Both have been used extensively to provide foundations for component-based software development, e.g. [11, 4]. For instance, advanced refinement and matching techniques [3] can be adapted for this context. Dynamic logic as an extension of Hoare logic can provide the framework. Design-by-contract is an important approach for the description of behavioural properties [12, 17, 8].

*Description logic* [2] relates to *knowledge engineering* – representing and reasoning about knowledge. The combination of knowledge and software engineering can result in fruitful outcomes.

# 5. CONCLUSIONS

Substantial effort is currently been made to make the Web a software development and deployment platform – the Web Services initiative. Component technology is ideally suited to support software development for the Web. However, this new application context also poses some challenges for component technology. Firstly, the Web is a ubiquitous platform, widely accepted and standardised. This requires component development techniques to adapt to this environment and to adhere to the standards of the Web. Secondly, the Web as a development platform is less well explored. As a consequence of distribution, decentralisation, and heterogeneity, the composition and assembly activities need to be well supported.

We have illustrated that technologies from another Web initiative – ontologies – can provide support for component-oriented Web service development. Ontology technology to represent and reason about knowledge can be adapted for components. We have explored the foundations for a composition framework for the Web. The framework is based on an ontology for components and services that incorporates reasoning support for behavioural aspects. An ontology – agreed and shared by developers and clients – can capture domain and software-technical knowledge.

Ultimately we aim to support flexible, collaborative, and adaptive component-based structures for the Web, ideally formed from federating, agent-like components. This would create an innovative, more autonomous software organisation. Of course, much work remains to be done until this vision is accomplished, but work also remains towards a fully implemented support environment. Aspects of automation will, if at all, be difficult to achieve.

# 6. REFERENCES

[1] A. Banerji et.al. *Web Services Conversation Language*. http://www.w3.org/TR/wscl10/, 2003.
[2] F. Baader, D. McGuiness, D. Nardi, and P. Schneider, editors. *The Description Logic Handbook*. Cambridge University Press, 2003.
[3] R. Back and J. von Wright. *The Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.
[4] A. Brogi, E. Pimentel, and A. Roldán. Compatibility of Linda-based Component Interfaces. In A. Brogi and E. Pimentel, editors, *Proc. ICALP Workshop on Formal Methods and Component Interaction*. Elsevier Electronic Notes in Theoretical Computer Science, 2002.
[5] F. Curbera, N. Mukhi, and S. Weerawarana. On the Emergence of a Web Services Component Model. In *Proc. 6th Int. Workshop on Component-Oriented Programming WCOP2001*, 2001.
[6] DAML-S Coalition. DAML-S: Web Services Description for the Semantic Web. In I. Horrocks and J. Hendler, editors, *Proc. First International Semantic Web Conference ISWC 2002*, LNCS 2342, pages 279–291. Springer-Verlag, 2002.
[7] D. Kozen and J. Tiuryn. Logics of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 789–840. Elsevier Science Publishers, 1990.
[8] G. Leavens and A. Baker. Enhancing the Pre- and Postcondition Technique for More Expressive Specifications. In R. France and B. Rumpe, editors, *Proceedings 2nd Int. Conference UML'99 - The Unified Modeling Language*. Springer Verlag, LNCS 1723, 1999.
[9] G. Leavens and M. Sitamaran. *Foundations of Component-Based Systems*. Cambridge University Press, 2000.
[10] F. Leymann. Web Services Flow Language (WSFL 1.0), 2001. www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf.
[11] M. Lumpe, F. Achermann, and O. Nierstrasz. A Formal Language for Composition. In G. Leavens and M. Sitamaran, editors, *Foundations of Component-Based Systems*, 2000.
[12] B. Meyer. Applying Design by Contract. *Computer*, pages 40–51, Oct. 1992.
[13] C. Pahl. A Formal Composition and Interaction Model for a Web Component Platform. In A. Brogi and E. Pimentel, editors, *Proc. ICALP Workshop on Formal Methods and Component Interaction*. Elsevier Electronic Notes in Theoretical Computer Science, 2002.
[14] C. Pahl. An Ontology for Software Component Matching. In *Proc. Fundamental Approaches to Software Engineering FASE'2003*. Springer-Verlag, LNCS Series, 2003.
[15] K. Schild. A Correspondence Theory for Terminological Logics: Preliminary Report. In *Proc. 12th Int. Joint Conference on Artificial Intelligence*. 1991.
[16] C. Szyperski. Component Technology - What, Where, and How? In *Proc. 25th International Conference on Software Engineering ICSE'03*, pages 684–693. 2003.
[17] J. Warmer and A. Kleppe. *The Object Constraint Language – Precise Modeling With UML*. Addison-Wesley, 1998.
[18] World Wide Web Consortium. *Web Initiaves*. www.w3.org, 2003.

# Modeling Multiple Aspects of Software Components

Roshanak Roshandel Nenad Medvidovic

Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781 U.S.A.
{roshande,neno}@usc.edu

## ABSTRACT

A software component is typically modeled from one or more of four functional aspects: interface, static behavior, dynamic behavior, and interaction protocol. Each of these aspects helps to ensure different levels of component compatibility and interoperability. Existing approaches to component modeling have either focused on only one of the aspects (e.g., interfaces in various IDLs) or on well-understood combinations of two of the aspects (e.g., interfaces and their associated pre- and post-conditions in static behavioral modeling approaches). This paper argues that, in order to accrue the true benefits of component-based software development, one may need to model all four aspects of components. However, this requires that consistency among the multiple aspects be maintained. We offer an approach to modeling components using the four-view perspective (called *the Quartet*) and identify the points at which the consistency among the models must be maintained.

## 1. INTRODUCTION

Component-based software engineering has emerged as an important discipline for developing large and complex software systems. Software components have become the primary abstraction level at which software development and evolution are carried out. We consider a software component to be any self-contained unit of functionality in a software system that exports its services via an interface, encapsulates the realization of those services, and possibly maintains transient internal state. In the context of this paper, we further focus on components for which information on their interface and behavior may be obtained. In order to ensure the desired properties of component-based systems (e.g., correctness, compatibility, interchangeability), both individual components and the resulting systems' architectural configurations must be modeled and analyzed.

The role of components as software systems' building blocks has been studied extensively in the area of software architectures [11,15]. In this paper, we focus on the components themselves. While there are many aspects of a software component worthy of careful study (e.g., modeling notations [2], implementation platforms [1], evolution mechanisms [9]), we restrict our study in this paper to an aspect only partially considered in existing literature: internal consistency among different models of a component. The direct motivation for this paper is our observation that there are four primary *functional* aspects of a software component: (1) *interface*, (2) *static behavior*, (3) *dynamic behavior*, and (4) *interaction protocol*. Each of these four aspects represents and helps to ensure different characteristics of a component. Moreover, the four aspects have complementary strengths and weaknesses. Existing approaches to component-based development typically select different subsets of these four aspects (e.g., interface and static behavior [9], or interface and interaction protocol [16]). At the same time, different approaches treat each individual aspect in very similar ways (e.g., modeling static behaviors via pre- and post-conditions, or modeling interaction protocols via finite state machines, or FSM).

The four aspects' complementary strengths and weaknesses, as well as their consistent treatment in literature suggest the possibility of using the four modeling aspects in concert. However, what is missing from this picture is an understanding of the different relationships among these different models in a single component. Figure 1 depicts the space of possible intra-component model relationship clusters. Each cluster represents a range of possible relationships, including not only "exact" matches, but also "relaxed" matches [17] between the models in question. Of these six clusters, only the pair-wise relationships between a component's interface and its other modeling aspects have been studied extensively (relationships *1*, *2*, and *3* in Figure 1).

This paper suggests an approach to completing the modeling space depicted in Figure 1. We discuss the extensions required to commonly used modeling approaches for each aspect in order to enable us to relate them and ensure their compatibility. We also discuss the advantages and drawbacks inherent in modeling all four aspects (referred to as the Quartet in the remainder of the paper) and six relationships shown in Figure 1. This paper represents a starting point in a much larger study. By addressing all the relationships shown in Figure 1 we eventually hope to accomplish several important long-term goals:

- enrich, and in some respects complete, the existing body of knowledge in component modeling and analysis,
- suggest constraints on and provide guidelines to practical modeling techniques, which typically select only a subset of the quartet,
- provide a basis for additional operations on components, such as retrieval, reuse, and interchange [17],
- suggest ways of creating one (possibly partial) model from another automatically, and
- provide better implementation generation capabilities from thus enriched system models.

The rest of the paper is organized as follows. Section 2 summarizes existing approaches to component modeling techniques and introduces the Quartet in more detail. Section 3 demonstrates our specific approach to component modeling using the Quartet and provides details of each modeling perspective. Section 4 discusses the relationships among the four modeling aspects shown in Figure 1 by identifying their interdependencies. Finally, Section 5 discusses our on-going research and future directions.

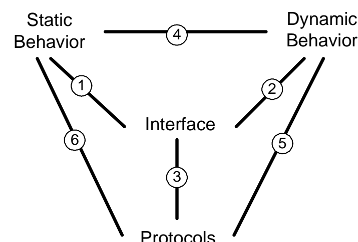



**Figure 1. Model relationships within a software component.**

# 2. COMPONENT MODELING

Modeling from multiple perspectives has been identified as an effective way to capture a variety of important properties of component-based software systems [2,3,6,8,10]. A well known example is UML, which employs nine diagrams (also called views) to model requirements, structural and behavioral design, deployment, and other aspects of a system. When several system aspects are modeled using different modeling views, inconsistencies may arise.

Ensuring consistency among heterogeneous models of a software system is a major software engineering challenge that has been studied in multiple approaches, with different foci. Due to space constraints, we discuss a small number of representative approaches here. [4] offers a model reconciliation technique particularly suited to requirements engineering. The assumption made by the technique is that the requirements specifications are captured formally. [5] also provide a formal solution to maintaining inter-model consistency, though more directly applicable at the software architectural level. One criticism that could be levied at these approaches is that their formality lessens the likelihood of their adoption. On the other hand, [7] provide more specialized approaches for maintaining consistency among UML diagrams. While their potential for wide adoption is aided by their focus on UML, these approaches may be ultimately harmed by UML's lack of formal semantics.

In this paper, we address similar problems to those cited above, but with a specific focus on multiple *functional* modeling aspects of a single software component. We advocate a four-level modeling technique (called *the Quartet*). Using the Quartet, a component's structural, behavioral (both static and dynamic), and interaction properties may be described and used in the analysis of a software system that encompasses the component:

1. *Interface* models specify the points by which a component interacts with other components in a system.
2. *Static behavior* models describe the functionality of a component discretely, i.e., at particular "snapshots" during the system's execution.
3. *Dynamic behavior* models provide a continuous view of *how* a component arrives at different states in its execution.
4. *Interaction protocol* models provide an *external* view of the component and its legal interactions with other components.

Typically, the static and dynamic component behaviors and interaction protocols are expressed in terms of a component's interface model (hence their positioning in Figure 1).

# 3. OUR APPROACH

The approach to component modeling we advocate is based on the concept of the Quartet discussed in the previous section: a complete functional model of a software component can be achieved only by focusing on all four aspects of the Quartet. At the same time, focusing on all four aspects has the potential to introduce certain problems (e.g., large number of modeling notations that developers have to master, model inconsistencies) that must be carefully addressed. While we use a particular notation in the discussion below, the approach is generic such that it can be easily adapted to other modeling notations. In this section, we focus on the conceptual elements of our approach, with limited focus on our specific notation used. Component models are specified from the following four modeling perspectives:[1]

```
Component_Model:
   (Interface, Static_Behavior,
    Dynamic_Behavior, Interaction_Protocol);
```

## 3.1. Interface

Interface modeling serves as the core of our component modeling approach and is extensively leveraged by other three modeling levels. An interface is specified in terms of several *interface elements*. Each interface element has a direction (+ or -), name (method signature), a set of input parameters, and possibly a return type (output parameter). The direction indicates whether the component *requires* (+) the service (i.e., operation) associated with the interface element or *provides* (-) it to the rest of the system. In other words:

```
Interface_Model: {Interface_Element};
Interface_Element:
   (Direction, Method_signature,
    {Input_parameter}, Output_parameter);
```

## 3.2. Static Behavior

We adopt a widely used approach to static modeling [9], which relies on first-order predicate logic to specify static behavioral properties of a component in terms of the component's *state variables*, the constraints associated with them (*invariants*), *interfaces* (as modeled in the interface model), *operations* (accessed via interfaces) and their corresponding *pre-* and *post-conditions.* In other words:

```
Static_Behaviors:
   ({State_variable}, Invariant, {Operation});
State_variable: (name, type);
Invariant: (logical_expression);
Operation:
   ({Interface_Element}, Pre_cond, Post_cond);
Pre/Post_cond: (logical_expression);
```

## 3.3. Dynamic Behavior

A dynamic behavior model provides a continuous view of the component's internal execution details. Variations of state-based modeling techniques have been typically used to model a component's internal behavior (e.g., in UML). Such approaches describe the component's dynamic behavior using a set of *sequencing constraints* that define legal ordering of the operations performed by the component. These operations may belong to one of two categories: (1) they may be directly related to the interfaces of the component as described in both interface and static behavioral models; or (2) they may be internal operations of the component (i.e., invisible to the rest of the system such as private methods in a UML class). To simplify our discussion, we only focus on the first case: publicly accessible operations. The second case may be reduced to the first one using the concept of *hierarchy* in StateCharts: internal operations may be abstracted away by building a higher-level state-machine that describes the dynamic behavior only in terms of the component's interfaces.

A dynamic model serves as a conceptual bridge between the component's protocol and static behavioral models. On the one hand, a dynamic model serves as a refinement of the static model as it further details a component's internal behavior. On the other hand, by leveraging a state-based notation, a dynamic model may be used to specify the sequence by which a component's operations get executed. Fully describing a component's dynamic behavior is essential in achieving two key objectives. First, it provides a rich model that can be used to perform sophisticated analysis and simulation of component behavior. Second, it can serve as an important intermediate level model to generate implementation level artifacts from architectural specification.

Existing approaches to dynamic behavior modeling employ an *abstract* notion of component state. These approaches treat states as entities of secondary importance, with the transitions between states playing the main role in behavioral modeling. Component states are often only specified by their name and set of incoming and outgoing transitions. We offer an extended notion of dynamic modeling that defines a state in terms of a set of variables maintained by the component and their associated invariants. These invariants constrain the values of and dependencies among the variables [14].

To summarize, our dynamic behavior model consists of a set of initial states and a sequence of guarded transitions from an origin to a destination state. Furthermore, a state is specified in terms of constraints it imposes over a set of a component's state

1. Concise formulations are used in this section to clarify our definitions and are not meant to serve as a formal specification of our model.

variables. In other words,

```
Dynamic_Behavior: (InitState,
   {State:(Direction)Transition->State});
State: (Name, Variables, Invariant);
Transition: (Label, {Parameter}, Guard);
Guard: (logical_expression);
```

### 3.4. Interaction Protocols

Finally, we adopt the widely used notation for specifying component interaction protocols, originally proposed in [11]. Finite state semantics are used to define valid sequences of invocations of component operations. Since interaction protocols are concerned with an "external" view of a component, valid sequences of invocations are specified irrespective of the component's internal state or the pre-conditions required for an operation's invocation. Our notation specifies protocols in terms of a set of initial states, and a sequence of transitions from an origin state to a destination.

```
Interaction_Protocol: (InitState,
   {State:(Direction)Transition->State});
State: (Name);
Transition: (Label, {Parameter});
```

## 4. RELATING COMPONENT MODELS

As previously discussed, modeling complex software systems from multiple perspectives is essential in capturing a multitude of structural, behavioral, and interaction properties of the system under development. The key issue however, is maintaining the consistency among these models [4,5,7]. We address the issue of consistency in the context of functional component modeling based on the Quartet technique.

In order to ensure the consistency among the models, their inter-relationships must be understood. Figure 1 depicts the conceptual relationships among these models. We categorize these relationships into two groups: syntactic and semantic. A *syntactic* relationship is one in which a model (re)uses the elements of another model directly and without the need for interpretation. For instance, interfaces and their input/output parameters (as specified in the interface model) are directly reused in the static behavior model of a component (relationship *1* in Figure 1). The same is true for relationships *2* and *3*, where the dynamic behavior and protocol models (re)use the names of the interface elements as transition labels in their respective FSMs.

Alternatively, a *semantic* relationship is one in which modeling elements are interpreted using the "meaning" of other elements. That is, specification of elements in one model *indirectly* affects the specification of elements in a different model. For instance, an operation's pre-condition in the static behavior model specifies the condition that must be satisfied in order for the operation to be executed. Similarly, in the dynamic behavior model, a transition's guard ensures that the transition will only be taken when the guard condition is satisfied. The relationship between a transition's guard in the dynamic behavior model and the corresponding operation's pre-condition in the static behavior model is semantic in nature: one must be interpreted in terms of the other (e.g., by establishing logical equivalence or implication) before their (in)consistency can be established. Examples of this type of relationship are relationships *4* and *5* in Figure 1.

In the remainder of this section we focus in more detail on the six relationships among the component model quartet depicted in Figure 1.

### 4.1. Interface vs. Other Models (Relationships *1*, *2*, *3*)

The interface model plays a central role in the design of other component models. Regardless of whether the goal of modeling is to design a component's interaction with the rest of the system or to model details of the component's internal behavior, interface models will be extensively leveraged.

When modeling a component's behaviors from a static perspective, the component's operations are specified in terms of interfaces through which they are accessed. As discussed in Section 3, an interface element specified in the interface model is mapped to an operation, which is further specified in terms of its pre- and post-conditions that must be satisfied, respectively, prior to and after the operation's invocation.

In the dynamic behavior and interaction protocol models, activations of transitions result in changes to the component's state. Activation of these transitions is caused by internal or external stimuli. Since invocation of component operations results in changes to the component's state, there is a relationship between these operations' invocations (accessed via interfaces) and the transitions' activations. The labels on these transitions (as defined in Section 3) directly relate to the interfaces captured in the interface model.

The relationship between the interface model and other models is syntactic in nature. The relationship is also unidirectional: all interface elements in an interface model may be leveraged in the dynamic and protocol models as transition labels; however, not all transition labels will necessarily relate to an interface element. Our (informal) discussion provides a conceptual view of this relationship and can be used as a framework to build automated analysis support to ensure the consistency among the interface and remaining three models within a component.

### 4.2. Static vs. Dynamic Behavior (Relationship *4*)

An important concept in relating static and dynamic behavior models is the notion of *state* in the dynamic model and its connection to the static specification of component's *state variables* and their associated *invariant*. Additionally, operation *pre- and post-conditions* in the static behavior model and *transition guards* in the dynamic behavior model are semantically related. We have extensively studied these relationships in [13,14] and identified the ranges of all such possible relationships. The corresponding concepts in the two models may be equivalent, or they may be related by logical implication. Although their equivalence ensures their inter-consistency, in some cases equivalence may be too restrictive. A discussion of such cases is given below.

**Transition Guard vs. Operation Pre-Condition.** At any state in a component's dynamic behavior model, multiple outgoing transitions may share the same label, but with different guards on the label. In order to relate an operation's pre-condition in the static model to the guards on the corresponding transitions in the dynamic model, we define the *union guard (UG)* of a transition label at a given state:

$$UG = \bigvee_{i=1}^{n} G_i$$

where $n$ is the number of outgoing transitions with the same label at a given state and $G_i$ is the guard associated with the ith transition.

Clearly, if UG is equivalent to its corresponding operation's pre-condition, the consistency at this level is achieved. However, if we consider the static behavior model to be an abstract specification of the component's semantics, the dynamic behavioral model becomes the concrete realization of those semantics. In that case, if UG is stronger than the corresponding operation's pre-condition, the operation may still be invoked safely: UG places bounds on the operation's (i.e., transition's) invocation, ensuring that the operation may never be invoked under circumstances that violate its pre-condition; in other words, UG should imply the corresponding operation's pre-condition.

**State Invariant vs. Component Invariant.** The state of a component in the static behavior specification is modeled using a set of state variables. The possible values of these variables are constrained by the *component's invariant*. Furthermore, a component's operations may modify the state variables' values, thus modifying the state of the component as a whole. The dynamic behavior model, in turn, specifies internal details of the component's states when the component's services are invoked. As described in Section 2, these states are defined using a name, a set of variables, and an invariant associated with these variables (called *state invariant*). It is crucial to define the states in the

dynamic behavior state machine in a manner consistent with the static specification of component's state and invariant.

Once again, an equivalence relation among these two elements may be too restrictive. In particular, if a state's invariant in the dynamic model is stronger than the component's invariant in the static model (i.e., state's invariant implies component's invariant), then the state is simply bounding the component's invariant, and does not permit for circumstances under which the component's invariant is violated. This relationship preserves the properties of the abstract specification (i.e., static model) in its concrete realization (i.e., dynamic model) and thus may be considered less restrictive than the equivalence. For more discussion on these, and a study of other possible relationships, see [14].

**State Invariants vs. Operation Post-Condition.** The final important relationship between a component's static and dynamic behavior models is that of an operation's post-condition and the invariant associated with the corresponding transition's destination state.

In the static behavior model, each operation's post-condition must hold true following the operation's invocation. In the dynamic behavior model, once a transition is taken, the state of the component changes from the transition's origin state to its destination state. Consequently, the state invariant constraining the destination state and the operation's post-condition are related. Again, the equivalence relationship may be unnecessarily restrictive. Analogously to the previous cases, if the invariant associated with a transition's destination state is stronger than the corresponding operation's post-condition (i.e., destination state's invariant implies the corresponding operation's post-condition), then the operation may still be invoked safely.

### 4.3. Dynamic Behavior vs. Protocol (Relationship *5*)

As previously mentioned, the relationship between the dynamic behavior and interaction protocol models of a component is semantic in nature: the concepts of the two models relate to each other in an indirect way.

As discussed in Section 3 we model a component's dynamic behavior by enhancing traditional FSMs with state invariants. Our approach to modeling interaction protocols also leverages FSMs to specify acceptable traces of execution of component services. The relationship between the dynamic behavior model and the interaction protocol model thus may be characterized in terms of the relationship between the two state machines. These two state machines are at different granularity levels however: the dynamic behavior model details the internal behavior of the component based on both internally- and externally-visible transitions, guards, and state invariants; on the other hand, the protocol model simply specifies the externally-visible behavior of the component, with an exclusive focus on transitions.

Our goal here is not to define a formal technique to ensure equivalence of two arbitrary state machines. This would first require some calibration on the models to even make them comparable. Additionally, several approaches have studied the equivalence of StateCharts [2,16]. Instead, we provide a more pragmatic approach to ensure the consistency of the two models. We consider the dynamic behavior model to be the concrete realization of the system under development, while the protocol of interaction provides a guideline for the correct execution sequence of the component's interfaces. Assuming that the interaction protocol model demonstrates *all* valid sequences of operations invocations of the component, the dynamic behavioral model should be designed such that any legal sequence of invocations of the component would also result in a legal execution of the component's dynamic behavior FSM. In other words, the dynamic behavioral model may be more general than the protocol of interactions; any execution trace obtained by the protocol model, must result in a legal execution of component's dynamic behavioral model.

### 4.4. Static Behavior vs. Protocol (Relationship *6*)

As discussed in Section 3.3, we consider the dynamic behavior model to be a bridge between a component's interaction protocol and static behavior specification models. The interaction protocol model specifies the valid sequence by which the component's interfaces may be accessed. In doing so, it fails to take into account the component's internal behavior (e.g., the pre-conditions that must be satisfied prior to an operation's invocation). Consequently, we believe that there is no direct conceptual relationship between the static behavior and interaction protocol models. Note, however, that the two models are related indirectly via a component's interface and dynamic behavior models.

## 5. CONCLUSION

In this paper, we argued for a four-level modeling approach, referred to as *the Quartet*, that can be used to model structural, static and dynamic behavioral, and interaction properties of a software component. We also discussed the conceptual dependencies among these models and highlighted specific points at which consistency among them must be established. While it may be argued that practitioners will be reluctant to use our approach in "real" development situations because it requires too much rigor and familiarity with too many notations, we believe such a criticism to be misplaced: the experience of UML has shown that practitioners will be all too happy to adopt multiple notations if those notations solve important problems. It should also be noted that our approach allows developers to select whatever subset of the Quartet they wish, but gives them an understanding of how incorporating additional component aspects is likely to impact their existing models.

## 6. REFERENCES


[1] Allen R., Garlan D., "A formal basis for architectural connection", *ACM TOSEM*, 6(3):213–249, 1997.

[2] Booch G., Jacobson I., Rumbaugh J. "*The Unified Modeling Language User Guide*", Addison-Wesley, Reading, MA.

[3] Farías A., Südholt M., "On Components with Explicit Protocols Satisfying a Notion of Correctness by Construction", in *Confederated Int'l Conf. CoopIS/DOA/ODBASE* 2002.

[4] Finkelstein A., et al., "Inconsistency Handling in Multi-Perspective Specifications", *IEEE TSE*, August 1994.

[5] Fradet P., et al., "Consistency checking for multiple view software architectures", in *ESEC/FSE 1999.*

[6] Hofmeister C., et al., "Describing Software Architecture with UML," in *WICSA1,* San Antonio, TX, February 1999.

[7] Hnatkowska B., et al., "Consistency Checking in UML Models", in *4th Int'l Conf. on Information System Modeling (ISM01),* Czech Republic, 2001.

[8] Krutchen, P.B. "The 4+1 View Model of Architecture", *IEEE Software 12*, pp. 42 - 50, 1995.

[9] Liskov B. H., Wing J. M., "A Behavioral Notion of Subtyping", *ACM TOSEM*, November 1994.

[10] Nuseibeh B., et al., "Expressing the Relationships Between Multiple Views in Requirements Specification", in *ICSE-15*, Baltimore, Maryland, 1993.

[11] Perry D.E., and Wolf A.L., "Foundations for the Study of Software Architectures", ACM SIGSOFT Software Engineering Notes, 17(4): 40-52, October 1992.

[12] Plasil F., Visnovsky S., "Behavior Protocols for Software Components", *IEEE TSE*, November 2002.

[13] Roshandel R., Medvidovic N., "Coupling Static and Dynamic Semantics in an Architecture Description Language", in *Working Conf. on Complex and Dynamic Systems Architectures,* Brisbane, Australia, December 2001.

[14] Roshandel R., Medvidovic N., "Relating Software Component Models", *Tech Rep't USC-CSE-2003-504*, March 2003.

[15] Shaw M., Garlan D., "Software Architecture: Perspectives on an Emerging Discipline", Prentice-Hall, 1996.

[16] Yellin D.M., Strom R.E., "Protocol Specifications and Component Adaptors," *ACM TOPLAS,* vol. 19, no. 2, 1997.

[17] Zaremski A.M., Wing J.M., "Specification Matching of Software Components", *ACM TOSEM*, vol. 6, no. 4, 1997.

# Reasoning About Parameterized Components with Dynamic Binding

Nigamanth Sridhar
Computer and Information Science
The Ohio State University
2015 Neil Ave
Columbus OH 43210-1277 USA
nsridhar@cis.ohio-state.edu

Bruce W. Weide
Computer and Information Science
The Ohio State University
2015 Neil Ave
Columbus OH 43210-1277 USA
weide@cis.ohio-state.edu

## ABSTRACT

Parameterized components provide an effective way of building scalable, reliable, flexible software. Techniques have been developed for reasoning about parameterized components in such a way that the relevant properties of a parameterized component can be predicted based on the *restrictions* on actual parameters. These techniques assume that the parameters are bound at compile-time. But in some cases, compile-time is just not late enough to instantiate a parameterized component; we would like to push instantiation into run-time instead. Doing this is sometimes dangerous, since we can no longer depend on the type system of the language to support our reasoning methods. In this paper, we present a specification notation and associated proof obligations, which when satisfied, allow us to extend the theories of reasoning about templates with static binding to dynamically-bound templates. We present these proof obligations in the context of the Service Facility pattern, which is a way of building templates whose parameters are dynamically bound.

## 1. INTRODUCTION

In languages that support them, templates can be used to program parameterized components, which can be specialized to meet specific client needs at component integration time. Further, since the language recognizes templates as first-class constructs, the compiler can enforce type restrictions on them as well as it does on other parts of the language. Reasoning about templates that are instantiated[1] at compile-time is considerably helped by the fact that each of the instantiated templates defines a new type that the compiler recognizes. Further, these types remain static for the rest of the program's lifetime. In general, a compiler that does compile-time template binding (*e.g.,* the C++ compiler) requires the following of the client program that uses a template:

**R1.** The template is instantiated statically, and

**R2.** The actual template parameters result in type-correct bodies for the template's methods.

One important consideration with parameterized components that could drastically affect their usefulness is the binding time of template parameters. If parameters are bound at compile-time (as in C++), we are faced with the problem that the component is statically configured, and no changes are possible after instantiation.

Fortunately, static composition is not inherent to parameterized programming [5]. The Service Facility (Serf) design pattern [10] provides a way of building parameterized software components, particularly in languages that do not provide linguistic support for templates. In contrast with C++ templates and Ada generics, which are instantiated at compile-time, template parameters are bound to a Serf at run-time. Such run-time binding has its advantages — the client has more flexibility in pushing design decisions to later in the program's lifetime [11]. However, late binding also has a downside — reasoning about program behavior becomes harder. Since Serf templates are instantiated at run-time, the reasoning system must be strengthened using additional proof obligations that subsume compile-time type checking and related checks, which we outline in this paper.

The rest of this paper is organized as follows. Section 2 reviews the Service Facility pattern, how to build parameterized components using this pattern, and the additional obligations that are needed to reason about the correctness of such components. We conclude in Section 3.

## 2. THE SERVICE FACILITY PATTERN

The Service Facility (Serf) design pattern [10] is a composite design pattern [8] that combines elements of several well-known design patterns [4], *viz.* Abstract Factory, Proxy, Factory Method, Bridge, and Strategy. Here we only describe the aspects of this pattern that enable parameterized programming. We refer the reader to [10] and to [9] for more details on other aspects of the pattern.

[1]In this paper we use the word *instantiation* to mean the setting of all parameters of a template. We refer to what is often called object instantiation in the OO literature as *object creation* in order to avoid confusion.

Poster presentation at the Workshop on Specification and Verification of Component-Based Systems, co-located with ESEC/FSE 2003. Sep 1–2, 2003. Helsinki, Finland.

Listing 1: C# Stack and StackSerf interfaces

```
1  public interface Stack : Data { }
2
3  public interface StackSerf : ServiceFacility {
4    void push(Stack s, Data x);
5    void pop(Stack s, Data x);
6    int length(Stack s);
7
8    // Template parameter(s)
9    public ServiceFacility ItemSerf { get; set; }
10 }
```

Listing 2: Instantiating StackSerf

```
1 /* ... */
2 PayrollRecordSerf pSerf = new PayrollRecordSerf_R1();
3 /* ... */
4 StackSerf stkSerf = new StackSerf_R1();
5 stkSerf.ItemSerf = pSerf;
6 /* ... */
```

When using the Serf design pattern, a client program dynamically supplies template parameters to the Serf template as *strategies*. Listing 1 shows the C# interface StackSerf, a stack template. A client program using this StackSerf will "instantiate" the template by assigning to the ItemSerf property a Serf isf that will provide the type of the item in the stack (a "strategy"). For example, Listing 2 shows a client instantiating implementation StackSerf_R1 of StackSerf to create a stack of payroll records.

Since the template parameters are set at run-time, there is no way for the compiler to ensure that they are set, let alone in a type-safe way (*i.e.,* with actuals that would have allowed compile-time type-checks to succeed). At the point that the stkSerf object is declared and constructed (line 4 in Listing 2), the compiler decides that the object is ready for use. However, under the semantics of Serfs, this object has not been fully instantiated and is therefore not ready for use. The client, therefore, has proof obligations that it has to satisfy — that the Serf has been properly instantiated with appropriate parameters (line 5).

The template parameters in a Serf are represented as data members in the implementation. Each template parameter corresponds to one (or two in some cases) data member in the Serf class. In order to ensure a Serf object has, in fact, been properly instantiated, the client has to satisfy a proof obligation that all of these data members have legal values. We will see later (Section 2.4) what such legal values are. In the rest of this section, we introduce new notation for specifying template parameters for Serfs.

## 2.1 Specifying Template Parameters

In order to specify a template Serf, we use the RESOLVE [2] notation. As an example, we present StackContract (borrowed from [2]) specified using the RESOLVE notation in Listing 3. This module defines one type (Stack) and its interface exports three operations on this type — push, pop, and length. The type definition describes a mathematical model (string of Item, in this case), as well as the set of legal values that a new instance of this type can assume upon initialization (empty string, in the case of Stack).

Listing 3: The contract for StackContract specified using RESOLVE

```
1  contract StackContract
2    context
3      global context
4        facility StandardIntegerFacility
5      parametric context
6        type Item
7    interface
8      type Stack is modeled by string of Item
9        exemplar s
10       initialization
11         ensures |s| = 0
12     operation push (
13         alters s: Stack,
14         consumes x: Item
15       )
16       ensures  s = <#x> * #s
17     operation pop (
18         alters s: Stack,
19         produces x: Item
20       )
21       requires |s| > 0
22       ensures  #s = <x> * s
23     operation length (
24         preserves s: Stack
25       ) : Integer
26       ensures  length = |s|
27 end StackContract
```

Each module can export zero or more types. In the case of a module that exports more that one type, the types are identified using a *type identifier*[2].

The *global context* of this contract introduces other modules or facilities that this component *uses*. In this particular example, StackContract makes use of an Integer component, and therefore *imports* the standard realization of that component (StandardIntegerFacility). In programming language terms, the global context serves the same purpose as Java import statements or C# using statements.

The parameters to this template are specified in its *parametric context*. In this example, StackContract is parameterized by the type of item that is contained in a stack. In general, parameters can be of four different kinds: constants, types, facilities, and math definitions. In this paper, we only deal with type and facility parameters, although the ideas can be easily extended to the other types of parameters as well. A *type* parameter lets the client specialize the template by supplying a specific type, as in our current example. A *facility* is an instance of some template. Thus, a facility parameter allows the client set up an integration-time relationship between components. The client can provide realizations of specific contracts that the template can use. Template parameters can also be *restricted* — the actual parameter could be required to implement certain functionality in a valid binding.

## 2.2 Specifying Serfs in RESOLVE

The contract in Listing 3 specifies that it requires, as part of its global context, the standard integer facility. Recall that a facility is an instance of a template, all of whose

[2]For the sake of simplicity, in this paper we only deal with Serfs that export exactly one type, and so we will no longer refer to the type identifier [10].

Listing 4: The contract for StackSerf

```
contract StackSerfContract
  context
    global context
      service facility StandardIntegerSerf
    parametric context
      service facility ItemSerf
        defining type Item
  interface
    type Stack is modeled by string of Item
      exemplar s
      initialization
        ensures |s| = 0
    operation push (
        alters s: Stack,
        consumes x: Item
      )
      ensures  s = <#x> * #s
    operation pop (
        alters s: Stack,
        produces x: Item
      )
      requires |s| > 0
      ensures  #s = <x> * s
    operation length (
        preserves s: Stack
      ) : Integer
      ensures  length = |s|
end StackSerfContract
```

formal parameters have been bound to actuals. In order to accommodate run-time binding of parameters, we introduce new notation to the RESOLVE language. A *service facility* is an instance of a template that is bound to its parameters dynamically, rather than statically.

Further, we unify all the different kinds of parameters that can be part of the parametric context of a RESOLVE template to be service facilities. In the case of type parameters, for instance, we specify in the parametric context a service facility that *defines* the required type.

Substituting service facilities for facilities, we can translate the RESOLVE StackContract (Listing 3) into StackSerfContract (Listing 4). It is easy to see that the Stack and StackSerf C# interfaces (Listing 1) can be generated from StackSerfContract. All the information needed to generate these interfaces is available in the contract. In general, the type exported by a SerfContract is used to generate the type interface (Stack in the example), and the interface part of the contract, along with the parametric context is used to generate the Serf interface.

## 2.3 Realizing RESOLVE Contracts as Serfs

Abstract RESOLVE components are expressed in the Serf approach as *interfaces*. In the languages that we consider (Java and .NET languages[3]), interfaces are first-class constructs in the language. Interfaces in these languages are comprised of *method signatures*. There is a direct mapping from the interface in the RESOLVE specification to the programming language interface. Further, the template parameters listed in the concept's parametric context are also represented by methods in the interface. Each facility parameter in the concept corresponds to two methods — one *setter*, and one *getter*[4]. For example, Listing 1 shows the C# interface for a StackSerf component.

## 2.4 Reasoning About Service Facilities

Now let us see how we can augment Serf interfaces with contract checking in order to enforce the proper use of Serfs as parameterized components. We will handle the two requirements, R1 and R2, separately.

*R1. Enforcing Instantiation.* Before the Serf can be used to create data objects, we require that the Serf has been properly instantiated, *i.e.,* all the template parameters have been set. For each template parameter that appears in the parametric context of the component, the data members that correspond to that parameter must have been set to values other than their initial values[5].

In order to make sure that by the time we use a Serf it is properly instantiated, we include a check to make sure that all the parameters have actually been set in the pre-condition of each method, including the create method. So, in accordance with design by contract [6], clients that want to use a Serf object have to first instantiate it by supplying appropriate actual parameters.

*R2. Enforcing Restrictions.* In the foregoing discussion, we have presented one way of making sure that a Serf is actually instantiated before it is used. However, how do we make sure that the parameters that have been supplied are appropriate from the type-checking standpoint?

To a limited extent, we can use the compiler to do these checks for us. In Listing 1, the method used to set ItemSerf takes a parameter of type ServiceFacility. So any legal (according to the C# compiler) invocation of this method should pass in a parameter that implements the ServiceFacility interface. This works, but only as long as we can bundle up all the restrictions on a particular template parameter into a single interface. But this is not possible in most cases. The following example illustrates this further.

Consider a Sort extension to StackSerf, StackSorterSerf. This component creates stacks that can be sorted[6]. For such a Serf, we have two separate restrictions on the ItemSerf parameter. First, this parameter, as in the regular StackSerf, should implement the ServiceFacility interface. Second, data objects created by this ItemSerf must be comparable to each other. That is, we should be able order these data objects according to some policy. Such a policy can be enforced by requiring this parameter to also implement the AreInOrder interface, presented in Listing 5.

A correct ItemSerf parameter to StackSorterSerf must implement both the ServiceFacility and AreInOrder interfaces. A naive way to enforce this using the C# compiler is to make AreInOrder extend ServiceFacility. Then, we can make

[3]All languages that respect the Common Type System of the Microsoft .NET Common Language Runtime have the same set of features [7]. Henceforth, whenever we want to refer to .NET languages, we will use C# as the representative.

[4]Again, we ignore the slight complication of allowing a single component to export multiple types.

[5]In this case, we will just use the initial value conventions of Java/C# — for example, Object type variables are initialized to be null, and int variables are initialized to 0.

[6]We do not care why someone would sort a stack. The purpose of this example is to illustrate problems with checking restrictions on parameters.

Listing 5: AreInOrder interface

```
1 public interface AreInOrder
2 {
3   public bool AreInOrder(Data x1, Data x2);
4 }
```

the type of the ItemSerf property StackSorterSerf to be AreInOrder. This way, the C# compiler could make sure that the parameter ItemSerf actually implements both interfaces. However, this solution is not desirable since in introduces a spurious inheritance relationship between ServiceFacility and AreInOrder where none really exists.

A better solution would be to create a new interface that extends both ServiceFacility and AreInOrder, and change the type of the ItemSerf() property such that it implements this new interface. While this solution works, and does not create bad inheritance hierarchies, it is cumbersome, requiring the creation of too many new interfaces.

Moreover, certain kinds of restrictions are semantic restrictions that cannot be enforced by the compiler. As an example, if a template takes two parameters that have to be related in some way, thee is no way for such a relation to to be encoded syntactically in C# (or Java).

The solution we advocate is again to rely on design by contract. We embed the restrictions on parameters in the specification of the component. From these specifications, we can then create instantiation-checking wrapper components that check whether a particular Serf has, in fact, been instantiated completely. These components are similar in spirit to checking components that ensure that the behavioral contract of the component is respected [3]. The difference here is that we check template instantiation.

Each of the parameters in the parametric context may be annotated with restrictions. For instance, in the StackSorterSerf example, we impose a restriction on the ItemSerf parameter that it implement the AreInOrder interface. This requirement, however, is stated in the RESOLVE contract, but not in the corresponding C# interface for reasons cited earlier in this section. Instead, the requirement is encoded as part of the instantiation-checking component. The setItemSerf method in the instantiation-checking wrapper for StackSorterSerf will now have a precondition that the parameter it gets passed implements the AreInOrder interface.

This precondition can be checked during execution using the Reflection API in C# and Java. The setItemSerf method uses reflection to query the parameter it gets passed to see the list of interfaces that parameter object implements. If AreInOrder is not part of this list, the check fails, and the instantiation does not complete successfully. This failure in instantiation is viewed as a failure to meet the contract, and is handled in the same way as in [1].

Apart from these, there are two more things that need to be checked as well. First, the object that is passed in as a parameter to the setItemSerf method is itself a Serf, and so we need to check if that Serf object has been properly instantiated. In order to perform this check, we include another method in the instantiation-checking wrapper that can be used to query if the Serf that is wrapped in it has been fully instantiated. This method returns a boolean value, after checking (locally) all of the parameters to the Serf. Second, in each method in the Serf, the parameters passed to the method must be checked to see if their runtime types match the expected type. For example, the parameter x to push in StackSerf must be of type Item.

The kinds of assertions that we are dealing with in checking instantiation are all actually checkable at run-time. They do not include arbitrary boolean predicates, but are very constrained — of the form "object implements a given interface", or "object's dynamic type is X", etc.

## 3. CONCLUSION

In this paper, we have presented a framework for reasoning about parameterized components whose parameters are bound at run-time. We have illustrated the use of this framework in the context of the Service Facility pattern, which is a design pattern that supports the construction of dynamically-bound parameterized components. The components are first specified using RESOLVE, and then realized as Serfs in an implementation language (C# in this paper). Further, we have outlined wrapper components that can check whether a particular Serf component has, in fact, been properly instantiated before it is used.

## 4. ACKNOWLEDGMENTS

This work has been supported by the National Science Foundation (NSF) under grant CCR-0081596, and by Lucent Technologies. Any opinions, findings, conclusions or recommendations expressed in this paper are those of the authors and do not reflect the views of the NSF or Lucent.

## 5. REFERENCES

[1] S. H. Edwards. Making the case for assertion checking wrappers. In *Proceedings of the RESOLVE Workshop 2002*, number Tech. Report TR-02-11, pages 28–42, Blacksburg, VA, June 2002.

[2] S. H. Edwards, W. D. Heym, T. J. Long, M. Sitaraman, and B. W. Weide. Specifying Components in RESOLVE. *SEN*, 19(4):29–39, 1994.

[3] S. H. Edwards, G. Shakir, M. Sitaraman, B. W. Weide, and J. Hollingsworth. A framework for detecting interface violations in component-based software. In *Proceedings: Fifth International Conference on Software Reuse*, pages 46–55, 1998.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1995.

[5] J. Goguen. Parameterized programming. *IEEE TSE*, SE–10(5):528–543, September 1984.

[6] B. Meyer. *Design by contract*, chapter 1. Prentice Hall, 1992.

[7] Microsoft. *Microsoft Visual C# .NET Language Reference*. Microsoft Press, Redmond, WA, 2002.

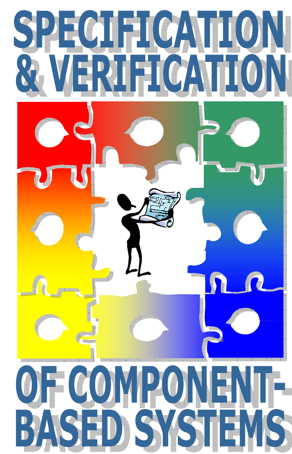
[8] D. Riehle. Composite design patterns.

[9] N. Sridhar, S. M. Pike, and B. W. Weide. Dynamic module replacement in distributed protocols. In *Proc. ICDCS-2003*, May 2003.

[10] N. Sridhar, B. W. Weide, and P. Bucci. Service facilities: Extending abstract factories to decouple advanced dependencies. In *Proc. ICSR-7*, pages 309–326, April 2002.

[11] H. Thimbleby. Delaying commitment. *IEEE Software*, 5(3):78–86, May/June 1988.

# SAVCBS 2003
# DEMONSTRATION ABSTRACTS



# Specifications in the Development Process: An AsmL Demonstration

Mike Barnett Colin Campbell Wolfgang Grieskamp Yuri Gurevich
Lev Nachmanson Wolfram Schulte Nikolai Tillmann Margus Veanes

Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399
USA
{mbarnett,colin,wrwg,gurevich,levnach,schulte,nikolait,margus}@microsoft.com

## ABSTRACT

AsmL is a specification system for software modeling, test generation, test validation, and implementation verification. It comprises a formal specification language, a set of libraries, and a test tool. It is integrated into the .NET Framework and Microsoft development tools. It has multiple source notations that can be used in a literate programming style either in an XML format or embedded within Microsoft Word. Specifications written in the system are executable, a novel feature which allows for semi-automatic test-case generation. In addition, the system can dynamically monitor an implementation to ensure that it conforms to its specification.

## 1. INTRODUCTION

There has been no lack of specification languages and systems for verifying properties of specifications and also for verifying that an implementation is correct with respect to its specification. What has been lacking (with several notable exceptions) are any results that have affected the "normal" programmer and the "normal" software development process. Our group, The Foundations of Software Engineering [4], is engaged in making specifications a part of the normal software development process at Microsoft. What we have found is that the area in most need of the kind of help specifications can provide is, perhaps surprisingly, testing and not development. Testers are often in the position of needing to understand the overall functionality of a system in order to design proper tests, yet this is often unavailable in any form other than the source code. A usable specification allows testing to begin earlier in the development process, concurrently with the coding effort. Currently, the natural language descriptions that serve as specifications suffer from ambiguity and imprecision. Finally, the most pressing need testers have, a *test oracle* can be provided by a specification.

We first describe the basics of AsmL specifications, then their use for test-case generation. A test-case consists of a sequence of calls to the modeled system, each call must be provided with a set of parameters. Then, when an implementation is available, the test-case can be applied to it and the results checked against its specification.

## 2. SPECIFICATIONS

AsmL is based upon the theory of Abstract State Machines (ASMs) [7, 8], which is a formal operational semantics for computational systems. A specification written in AsmL is an operational semantics expressed at an arbitrary level of abstraction. We call such an operational semantics a *model program*; we use the terms model and specification interchangably. AsmL incorporates the following features:

**Nondeterminism** AsmL provides a carefully chosen set of constructs with which one can express nondeterminism. They allow the specification of a range of behaviors within which the implementation must remain. Overspecification can be avoided without sacrificing precision.

**Transactions** AsmL is inherently parallel: all assignment statements are evaluated in the same state and all of the generated updates are committed in one atomic transaction. Updates that must be made sequentially are organized into *steps*; the updates in one step are visible in following steps and steps can also be organized hierarchically. Limiting the number of steps to prevent unnecessary sequentialization also helps prevent overspecification. The next state of the component is fully specified without making implementation-level decisions on how to effect the changes.

Additionally, AsmL provides a rich set of mathematical data types, such as sets, sequences, and maps (finite functions), along with advanced programming features such as pattern matching and several types of comprehensions.

It is also a full .NET language; AsmL models can interoperate with any other .NET component, e.g., written in

$C^\sharp$, VB, or C++. There are two source notations, a VB-like style in which white space is used to indicate scoping and which looks very similar to pseudo-code, and another style which is a superset of $C^\sharp$. Both notations can be used within a literate programming system; AsmL models are embedded in Word documents where they appear in special style-blocks. AsmL models can be also authored from Visual Studio .NET, where they are represented as XML documents in a particular schema. Bi-directional conversion between XML and Word format is supported. AsmL models can be compiled and directly executed from within Word or Visual Studio. AsmL specifications can be made at the interface level or for individual classes. More details on the use of AsmL specifications can be found in several papers [1, 6].

# 3. PARAMETER SELECTION

Our method for selecting test-case parameters is based on Korat [3]. The user adds annotations to the model to describe possible values for types and method parameters. From these annotations, the tool derives parameter sets. The annotations consist of values associated with parameters and fields of basic types, as well as a set of predicates that serve as filters on the cross-product of the combinations of parameters and define invariants for complex types. A technique called access driven filtering is used to enumerate the parameter sets in an efficient way that avoids generating redundant combinations.

# 4. SEQUENCE SELECTION

The tool exhaustively explores the reachable state space of the model, by executing methods with all associated parameters [5]. By necessity, the exploration must be pruned to some finite bound. We utilize a variety of cooperating techniques. One technique is a bound for the branch coverage. Another one can be considered as "state space" coverage, and is based on grouping the states (variable bindings) of the model into equivalence classes and bounding the number of representatives visited during exploration for each equivalence class. Finally, direct filtering of states can also be indicated; any state violating a filter is discarded and not considered as part of the reachable state space.

# 5. CONFORMANCE CHECKING

The use of AsmL specifications for conformance checking has been described elsewhere [2]. Conceptually, we run the specification and the implementation in parallel and check that the behavior of the latter is a possible behavior of the former. We track objects as they are created, returned from an implementation and then have their instance methods called. In this regard, the specification functions as a test oracle; this increases the efficacy of testing. Arbitrary conformance relations can be specified between the state of the model and the state of the implementation to allow a finer-grain checking than just comparing return values.

The conformance checking works both for test sequences derived from the model and those that are externally supplied. The implementation is instrumented at the IL level (the platform-independent language of the .NET virtual machine); this allows any .NET implementation, irrespective of its source language, to be checked relative to an AsmL specification.

# 6. CONCLUSION

One crucial decision made in the development of AsmL was to not limit its expressiveness. This has the consequence that general AsmL models are not directly amenable to static verification, such as model checking. However, we are investigating several methods for enforcing restrictions to enable at least some static verification.

We also are continuing to refine the source notations and the test tool itself in close collaboration with several product groups. We strongly believe that the timing is right for formal specifications to become an integral part of industrial software development.

# Acknowledgements

This work would not have been possible without the efforts of visiting researchers and interns that have spent time in the Foundations of Software Engineering group at Microsoft Research. We also are indebted to the product groups that have worked with us and provided valuable feedback.

# 7. REFERENCES

[1] M. Barnett and W. Schulte. The ABCs of specification: AsmL, behavior, and components. *Informatica*, 25(4):517–526, Nov. 2001.

[2] M. Barnett and W. Schulte. Runtime verification of .NET contracts. *Journal of Systems and Software*, 65(3):199–208, 2003.

[3] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. *Software Engineering Notes*, 27(4), 2002.

[4] Foundations of Software Engineering, Microsoft Research, 2003. `http://research.microsoft.com/fse`.

[5] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. *Software Engineering Notes*, 27(4):112–122, 2002. From the conference International Symposium on Software Testing and Analysis (ISSTA) 2002.

[6] W. Grieskamp, M. Lepper, W. Schulte, and N. Tillmann. Testable use cases in the abstract state machine language. In *Asia-Pacific Conference on Quality Software (APAQS'01)*, Dec. 2001.

[7] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[8] Y. Gurevich. Sequential Abstract State Machines Capture Sequential Algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, July 2000.

# Mae
# An Architectural Evolution Environment

Roshanak Roshandel

Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781 U.S.A.
roshande@usc.edu

## ABSTRACT

We present Mae an architectural evolution environment, built upon a system model that combines architectural and configuration management concepts into a single representation. Through Mae, users can specify architectures (in terms of their constituent components, connectors, and interfaces) in a traditional manner, manage the evolution of the architectures using a check-out/check-in mechanism that tracks all changes, select a specific architectural configuration, and analyze the consistency of a selected configuration.

## 1. ARCHITECTURAL CHANGE

Consider the following scenario. An organization specializing in software development for mobile platforms is commissioned by a local fire department to produce an innovative application for "on the fly" deployment of personnel in situations such as natural disasters and search-and-rescue efforts. Following good software engineering practices, the organization first develops a proper architecture for the application in a suitable architectural style, then models this architecture in an architecture description language (ADL), refines the architecture into a module design, and, finally, implements the application impeccably. The new application is an instant hit, and fire and police departments across the country adopt it. Motivated by this success, as well as by demands for similar capabilities from the military, the organization enters a cycle of rapidly advancing the application, creating add-ons, selling upgrades, adapting the application to different hardware platforms (both stationary and mobile), specializing the application for its various customers, and generally increasing its revenue throughout this process.

Configuration management (CM) systems have long been used to provide support for these kinds of situations. This, however, leads to problems with the above scenario: as the application evolves, so does its architecture. These architectural changes must be managed in a manner much like source code, allowing the architecture to evolve into different versions and exhibit different variants. One solution is to store the entire architectural description in a single file and track its evolution using an existing CM system (called *coarse-grained versioning*). An alternative solution is to version each architectural element in a separate file (called *fine-grained versioning*). Problems associated with each of these approaches reduce their effectiveness in managing architectural evolution. These problems are briefly discussed in Section 2.

Any solution to managing architectural evolution must support an architect in using: 1) multiple versions of a single architectural element that are part of the same configuration, 2) optional elements, 3) variant elements, 4) elements that are both optional and variant, and 5) relations among optional and variant elements. To address these issues and to mitigate the problems associated with use of traditional CM systems, we have developed a novel approach called Mae. Mae combines techniques from the fields of software architecture and configuration management to make two unique contributions: 1) an architectural system model that facilitates capturing the evolution of an architecture and its constituent elements, and 2) an integrated environment that supports managing the evolution of architectures. Details of the system model may be found in [2]. We propose to demonstrate Mae's architectural evolution environment and its functionality in *designing*, *analyzing*, and *evolving* software architectures.

## 2. EXISTING APPROACHES

Even though it is possible to manage the evolution of the architectural artifacts using traditional CM systems, we argue that this cannot be done effectively.

### 2.1. Coarse-grained Versioning

One possible approach to using an existing CM system for managing architectural evolution is to store and version the entire architectural description as a single file. This solution is akin to storing and versioning the entire source code of a software program as a single file. Clearly, managing artifacts at such a coarse-grained level leads to severe problems since any single change would result in a new version of the entire architectural specification. Moreover, the presence of multiple optional and variant elements leads to a combinatorial explosion of branches, due to the fact that each potential combination must be explicitly specified. Finally, this approach prevents the use of multiple versions of the same artifact within a single architecture. In sum, these shortcomings make versioning an entire architectural specification as a single artifact a highly undesirable solution for managing architectural evolution.

### 2.2. Fine-grained Versioning

Versioning fine-grained artifacts is considered a better approach to managing source code evolution than coarse-grained versioning. However, the analogy does not hold when applied to architectural evolution. Fine-grained versioning leads to serious consistency problems due to the fact that the architectural specification and the CM system capture duplicate information about the architectural configuration. Any change in the composition of the architectural configuration must be reflected in the CM system, and vice versa. Given that much of architectural design resolves around composing an architectural configuration, this becomes a recurrent and potentially error-prone activity.

This approach also requires extensive use of branching to manage optionality and variability. Traditional CM techniques that support branching (e.g., differencing and merging) work well for source code. However, they simply do not work for

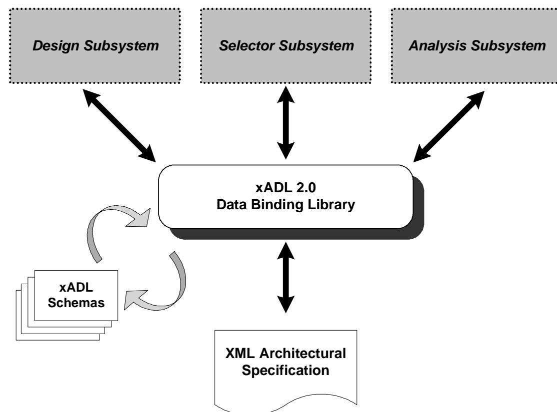



**Figure 1. Mae's Architecture**

architectural specifications, because of a difference in the level of granularity (i.e., lines of code in a source file versus components, connectors, links, and so on in an architectural specification). As a result, using a traditional CM system would force an architect into storing each potential architectural configuration on a separate branch. Finally, this approach requires breaking up an architectural specification into numerous small files to be managed separately. Even for a medium-sized application, this results in hundreds of small files that must be managed. While automated tools could be created to address this problem, the aforementioned problem of keeping the architectural specification and the CM system synchronized remains a significant obstacle.

To summarize, neither coarse-grained nor fine-grained versioning provides an adequate solution for capturing and managing architectural evolution. We have developed a system model that addresses the above issues. The system model captures architecture in terms of types and instances of constituent components, connectors, and their interfaces; leverages behaviors, constraints, and subtyping relationships among them; employs revisions and inter-file branches to support linear and diverging paths of evolution; and uses guarded expressions to denote optionality and variability of the artifacts. Finally it supports hierarchical composition of components and connectors in the system. The full discussion may be found in [2].

In the next section we present our architectural evolution environment, which relies on the above system model, and addresses the architectural evolution problem.

# 3. MAE ENVIRONMENT

Mae's architecture evolution environment provides and enforces the specific procedures through which an architecture is created and evolved. The environment does so by providing a tightly-integrated combination of functionality that covers both architectural aspects, such as designing and specifying an architecture or analyzing an architecture for its consistency, and CM aspects, such as checking out and checking in elements that need to change or selecting a particular architectural configuration out of the available version space.

As shown in Figure 1, the Mae architectural evolution environment consists of four major subsystems. The first subsystem, the xADL 2.0 data binding library [1], forms the core of the environment. The data binding library is a standard part of the xADL 2.0 infrastructure that, given a set of XML schemas, provides a programmatic interface to access XML documents adhering to those schemas. In our case, the data binding library provides access to XML documents described by the XML schemas that represent Mae's integrated system model. In essence, thus, the xADL 2.0 data binding library encapsulates our system model by providing a programmatic interface to access, manipulate, and store evolving architecture specifications.

The three remaining subsystems each perform separate but complimentary tasks as part of the overall process of managing the evolution of a software architecture:

- The design subsystem combines functionality for graphically designing and editing an architecture with functionality for versioning the architectural elements. This subsystem supports architects in performing their day-to-day job of defining and maintaining architectural descriptions, while also providing them with the familiar check out/check in mechanism to create a historical archive of all changes they make.
- The selector subsystem enables a user to select one or more architectural configurations out of the available version space. Once an architecture has started to evolve, and once it contains a multitude of optional and variant elements, the burden of manually selecting an architectural configuration becomes too great. To overcome this burden and automatically extract a single architecture based upon a user-specified set of desired properties, Mae provides the subsystem as an integral part of its environment.
- Finally, the analysis subsystem provides sophisticated analyses for detecting inconsistencies in architectural configurations. This subsystem typically is used after a particular architectural configuration has been selected, and helps to ensure that the architectural configuration is not only structurally sound, but also consistent with the expected behaviors and constraints of each and every component and connector in the selected configuration.

# 4. EVALUATION

Mae has been successfully used in three different settings as the primary architectural development and evolution environment. The collective experiences not only show that Mae is effective in circumventing the problems that occur when using a traditional CM system, but also demonstrate that it is a usable and scalable solution that is applicable to real-world problems.

As a first experience, we used Mae to create and evolve the architecture of an audio/video entertainment system patterned after an existing architecture for consumer electronics. Our evaluation focused on usability, and in particular on whether the presence of configuration management functionality hinders or obscures the process of designing an architecture. Our second experience with Mae involved creating and evolving the software architecture of the Troops Deployment and battle Simulations system. The evaluation focused on evaluating the scalability of Mae. While the system contains a moderate number of component and connector types, the number of component and connector instances can be in the 100's. Finally, we evaluated Mae's applicability to real-world settings through independent use by another research group at the University of Southern California. This group collaborates with NASA's Jet Propulsion Laboratory (JPL) in modeling and analyzing the evolving software architecture of the SCrover application, which is the on-board software of a rover system built using JPL's Mission Data System (MDS) framework

# 5. REFERENCES


[1] Dashofy, E.M., van der Hoek. A., Taylor R.N., An Infrastructure for the Rapid Development of XML-based Architecture Description Languages, in *Proceedings of the 24th International Conference on Software Engineering (ICSE2002)*, Orlando, Florida.

[2] Roshandel R., van der Hoek A., Mikic-Rakic M., Medvidovic N., Mae - A System Model and Environment for Managing Architectural Evolution, *Submitted to ACM Transactions on Software Engineering and Methodology (In review)*, October 2002.

# Runtime Assertion Checking Using JML

Roy Patrick Tan
Department of Computer Science
Virginia Tech
660 McBryde Hall, Mail Stop 0106
Blacksburg, VA 24061, USA
rtan@vt.edu

```
public class IntMathOps3 {

  //@ requires y >= 0;
  public static int isqrt(int y)
  {
    return (int) Math.sqrt(y);
  }
}
```

**Figure 1: A simple specification requiring the parameter to be non-negative**

## Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*programming by contract, assertion checkers, class invariants*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*pre- and post-conditions, invariants, assertions*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*object-oriented programming*; D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids*; D.3.2 [**Programming Languages**]: Language Classifications—*JML*

## General Terms

Languages

## Keywords

JML, Java, run-time checking, design by contract

## 1. INTRODUCTION

JML, the Java Modeling Language, is a language that allows programmers to specify the detailed design of Java programs. A software developer can use JML to add specifications such as method preconditions and postconditions, and class invariants to clearly indicate their correct behavior.

*SAVCBS* '03 Helsinki, Finland

```
//@ model import org.jmlspecs.models.*;

public class IntMathOps2 {

 /*@ public normal_behavior
   @   requires y >= 0;
   @   assignable \nothing;
   @   ensures -y <= \result && \result <= y;
   @   ensures \result * \result <= y;
   @   ensures
   @        y < (Math.abs(\result) + 1)
   @            * (Math.abs(\result) + 1);
   @*/
  public static int isqrt(int y);

}
```

**Figure 2: A complete specification**

JML annotations are written in the form of specially commented sections of the code. Figure 1 shows a lightweight JML specification for an integer square root method, requiring that the input be non-negative [4]. While specifications can be as simple as that, JML has sophisticated features that allow programmers to write full, abstract, model-based specifications. Figure 2 shows a complete specification for the integer square root method [4].

## 2. THE JML COMPILER

Jmlc, the JML compiler, is one of several tools support the JML notation [2]. Jmlc takes JML annotated source files and compiles preconditions, postconditions, invariants, and history constraints into bytecode that checks these specifications at runtime, making jmlc an ideal design-by-contract tool.

Runtime monitoring of contract assertions has many well known advantages. One particularly useful feature of runtime assertion checking is that an error is likely to be found at the point of contract violation; the error does not propagate such that when it is detected, the point of failure is in correctly implemented code.

The lack of assertions in early versions of the Java language, and the lack of design-by-contract checking of preconditions, prostconditions, and invariants in the current version (JDK

1.4) has led to many runtime assertion checking tools for Java, such as iContract [3], and Jass [1].

However, JML has features not found in other runtime checkers. For example, JML's facility for specification-only fields and methods allows programmers to create specifications using an abstract model of the object's state. The JML compiler allows programmers to use formal specification as a practical tool for debugging and testing software components.

## 3. CONCLUSIONS

Any developer of Java components and applications may benefit from the use of JML tools. JML has an easy adoption path, since classes and methods need not have full specifications. The programmer can begin by putting the odd precondition or postcondition check and then code in more complex specifications as his proficiency in the language improves. JML has been put to practical use in industry. Particularly, nearly all the API of Java Card, a dialect of Java for use in smart cards, has been specified in JML [5].

JML was originally developed in Iowa State University by Gary Leavens and his students. It is now an open source project with developers from all over the world actively contributing to improve the tools and the language. The JML homepage can be found at http://jmlspecs.org.

## Acknowlegements

We gratefully acknowledge the financial support from the National Science Foundation under the grant CCR-0113181. Any opinions, conclustions or recommendations expressed in this paper do not necessarily reflect the views of the NSF.

## 4. REFERENCES

[1] D. Bartetzko, C. Fischer, M. Mller, and H. Wehrheim. Jass - java with assertions. In K. Havelund and G. Rosu, editors, *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier, 2001.

[2] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. Rustan, M. Leino, and E. Poll. An overview of JML tools and applications. In *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS '03)*, volume 80, pages 73–89. Elsevier, 2003.

[3] R. Kramer. iContract — the Java design by contract tool. In *TOOLS 26: Technology of Object-Oriented Languages and Systems*, pages 295–307, 1998.

[4] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06v, Department of Computer Science, Iowa State University, May 2003.

[5] E. Poll, J. van den Berg, and B. Jacobs. Specification of the JavaCard API in JML. In J. Domingo-Ferrer, D. Chan, and A. Watson, editors, *Fourth Smart Card Research and Advanced Application Conference (CARDIS'2000)*, pages 135–154. Kluwer Acad. Publ., 2000.