Journal of Software Engineering for Robotics 6(1), December 2015, 15-32
ISSN: 2035-3928

An Architecture-oriented Approach to System Integration in Collaborative Robotics Research Projects — An Experience Report

David VERNON[1,*] Erik BILLING[1] Paul HEMEREN[1] Serge THILL[1] Tom ZIEMKE[1,2]

[1] Interaction Lab, School of Informatics, University of Skövde, Sweden
[2] Human-Centred Systems, Department of Computer and Information Science, Linköping University, Sweden

Abstract—Effective system integration requires strict adherence to strong software engineering standards, a practice not much favoured in many collaborative research projects. We argue that component-based software engineering (CBSE) provides a way to overcome this problem because it provides flexibility for developers while requiring the adoption of only a modest number of software engineering practices. This focus on integration complements software re-use, the more usual motivation for adopting CBSE. We illustrate our argument by showing how a large-scale system architecture for an application in the domain of robot-enhanced therapy for children with autism spectrum disorder (ASD) has been implemented. We highlight the manner in which the integration process is facilitated by the architecture implementation of a set of placeholder components that comprise stubs for all functional primitives, as well as the complete implementation of all inter-component communications. We focus on the component-port-connector meta-model and show that the YARP robot platform is a well-matched middleware framework for the implementation of this model. To facilitate the validation of port-connector communication, we configure the initial placeholder implementation of the system architecture as a discrete event simulation and control the invocation of each component's stub primitives probabilistically. This allows the system integrator to adjust the rate of inter-component communication while respecting its asynchronous and concurrent character. Also, individual ports and connectors can be periodically selected as the simulator cycles through each primitive in each sub-system component. This ability to control the rate of connector communication considerably eases the task of validating component-port-connector behaviour in a large system. Ultimately, over and above its well-accepted benefits for software re-use in robotics, CBSE strikes a good balance between software engineering best practice and the socio-technical problem of managing effective integration in collaborative robotics research projects.

Index Terms—best practice in robotics, model-driven engineering, component-based software engineering, discrete event simulation, YARP, component-port-connector model.

1 Introduction

Collaborative research projects, interdisciplinary ones in particular, tend not to adopt industry-strength software engineering practices because they aim to develop proof-of-principle prototypes rather than commercial products, because of the high overhead in effort, and because most research projects are funded for scientific results and not for infrastructure and integration [1], [2]. Nevertheless, where large software systems are concerned, even proof-of-principle implementations require a strong element of good engineering practice if they are to be successful. A balance is needed between the rigour demanded by industry standards (and contemporary software engineering practice) and the flexibility required to foster voluntary collaboration. This balance becomes particularly problematic when software developed by geographically-distributed teams has to be integrated into a cohesive system.

Part of the solution to this problem lies in the realization that "integration should *not* start at the level of software code, but at the level of *models* of the provided functionality" [3], e.g. the system architecture. In this paper, based on our experience in the DREAM research project (see Section 2), we argue that an effective balance between rigour and flexibility can

Regular paper – *Manuscript received August 16, 2015; revised December 04, 2015.*

- *This work was supported by the European Commission, Project 611391: DREAM — Development of Robot-enhanced Therapy for Children with Autism Spectrum Disorders.*

be struck by taking an operational-architecture[1] approach to system integration and adopting component-based software engineering (CBSE) [4], [5], [6], [7]. While CBSE is normally exploited to promote greater software re-use, here we wish to highlight the advantages it offers for system integration, realized through its focus on component composability.

The other part of the solution to the problem of achieving the right balance lies a second realization that integration is a socio-technical issue concerned with project management in circumstances involving independently-minded researchers that are not always inclined to adopt extensive strict standards. CBSE offers something much better than a least-worst resolution of this project management and software integration dilemma: sound engineering practice with minimal overhead while affording considerable flexibility for the developer.

Thus, the main purpose of this article is not to propose a particular architecture nor even a tool for designing an architecture, but a methodology that makes the process of software development and subsequent integration more effective and efficient in collaborative research projects. It is well acknowledged that any large system requires an architecture to be designed first and that this architecture drives the subsequent software design and development process. Our central point is that this architecture can be *operational* first and that by so doing it becomes a driver for the software development of each constituent sub-system. This facilitates integration by providing hard constraints on the outcome of the software sub-system development without the imposition of hard constraints on all the other phases of the software sub-system development lifecycle. Thus, the contribution of the paper is primarily methodological, with support for that methodology being provided by a discrete event simulation technique to facilitate handling the complexity of integration that naturally arises from a complete implementation of the full system architecture and the large number of associated inter-sub-system communications. It is a relatively straightforward approach that provides clear integration acceptance criteria, substantially easing the burden on the developers without compromising the integration process. This results in a win-win situation for the developers and the system integrator.

Having said that, software development and integration does not take place in a vacuum and the implemented operational architecture, considered as a driver for both sub-system development and integration, is not sufficient by itself. There are two other key elements: the user requirements and system specification that define the software functionality, and the software engineering process by which the required software is developed.

1. The phrase *operational architecture* is sometimes used to refer to a way of structuring complex operations, often in the military domain where it is one of three views on the enterprise architecture. However, in this paper we use it in a different way: to emphasize that the architecture itself is realised as a working system of interacting components, each of which acts as a placeholder for software that is subsequently integrated.

We deal with the system specification and its encapsulation in the system architecture in Section 6, highlighting its grounding in requirements derived from use cases in order to ensure that the inter-sub-system interfaces capture the semantics of the information exchanged.

We deal with the software standards in Section 7. Here it is important to note that the processes of both software development and integration are governed by standards covering the full development lifecycle. These standards form the essential backdrop for the software development and integration effort. It is significant that the approach advocated in the paper allows two classes of standards to be adopted: *mandatory standards* and *recommended standards*. The mandatory standards relate to those aspects of the software that refer to integration while the recommended standards relate to everything necessary for that sub-system to achieve the required functionality.[2] Significantly, each team of developers responsible for the three functional architecture sub-system components has developed their own specific sub-system architecture using their own preferred modeling, specification, design, implementation, and test practices. The only constraints on their sub-system architecture design derive from the standards imposed by mandatory standards. How they achieve this is completely up to them. Although the recommended standards do provide some advice, the developers are under no obligation to take it. This is the win-win that derives from the operational-architecture approach described in the paper: freedom to adopt or not industrial-strength standards governing the full software development life-cycle and unproblematic integration provided they adhere to the mandatory standards.

Within the broad domain of CBSE, we focus in particular on the component-port-connector (CPC) meta-model, i.e, level two of the Object Management Group's (OMG) four-levels of abstraction in model-driven engineering (MDE) [8]. Effective implementation of the CPC meta-model imposes a minimal number of necessary software engineering practices. We show how one specific development platform, YARP [9], [10], provides a suitable middleware framework to support these practices. The operational-architecture approach suggested here provides additional flexibility for developers by only specifying the architecture at the sub-system levels, allowing each sub-system to then take advantage of the emergent architecture approach [2], allowing the sub-system to be developed as required without compromising the standards required for effective integration.

We demonstate the approach by showing how the sys-

2. As we will see in Section 4, the mandatory standards relating to those aspects of the software that refer to integration, i.e. the standards imposed by system architecture component specification, correspond to three of the four Cs of component-based software engineering: coordination, configuration, and communication. The recommended standards relate to the fourth C: computation, i.e. everything necessary for that sub-system to achieve the required functionality and cover the full software development lifecycle. Some minimal mandatory standards were in fact adopted for computation but these related only to the manner in which the functionality is encapsulated.

tem architecture for a humanoid robotics application in the domain of robot-enhanced therapy for children with autism spectrum disorder (ASD) has been implemented. In particular, we highlight the advantage of the component-based approach for system integration by realizing the complete system architecture as four stub-driver sub-systems implemented as four distinct placeholder components. These four components realize all the communciation ports and connectors for the complete system. Since the number of ports and connectors is large (of the order of one hundred in either case), it is not trivial to ensure that communication over all these connectors is working correctly, especially because all communciation takes place asynchronously and concurrently. On the other hand, once it has been established that inter-component communication is working effectively, subsequent integration of functional software is made much easier when replacing stub and driver primitives with the operational primitives. To solve the problem of validating all inter-component communication, we implement the system architecture as a discrete event simulation and control the invocation of each component driver probabilistically. This allows the system integrator to adjust the rate of inter-component communication during integration while not violating its asynchronous and concurrent character. Furthermore, individual ports and connectors can be periodically selected as the simulator cycles through each primitive in each sub-system component. This ability to control the connector communication considerably eases the task of validating component-port-connector behaviour.

2 The DREAM Project

DREAM [11] is a $4^1/_2$-year European project in which seven partners are involved in developing software for robot-enhanced therapy for children with ASD. Social robots — specifically the Nao [12] and Probo [13] robots — are used as tools to help these children develop imitation, joint attention and turn-taking skills, with the final objective of achieving better real-life human-human social interaction.

Researchers have found that the use of technological tools improves the acquisition of social skills in children with ASD [12], [13], [14], [15]. In particular, robots might have the potential to act as intermediaries between human therapists and patients (see [16]). There are three ways for this to be accomplished: the robo-therapist, the robo-assistant, and the robo-mediator approaches. In the robo-therapist approach, the robot acts by itself as the therapist and completely replaces the human agent. In the robo-assistant approach, the robot acts as a facilitator of the therapy intervention but is not necessary for successful treatment and could be replaced by another agent, e.g., an animal. In the robo-mediator approach, the robot is used as a means for delivering the treatment because it enables faster and better gains from the therapeutic intervention.

DREAM constitutes an extension of the robo-mediator approach (often referred to has robot-assisted therapy or RAT) in which the robot is a necessary component of the therapy, forming a third element in the therapy intervention alongside the child and the therapist, but the robot has greater autonomy in conducting the intervention [17]. The robot's behaviour is still supervised by the therapist and, consequently, it exhibits a form of autonomy referred to as *supervised* or *shared* autonomy. We refer to this next-generation RAT as *robot-enhanced therapy* or RET. To achieve this level of autonomy, while following the intervention script, the robot must be able to (a) gather sensory data about the child, (b) assess psychological disposition and behaviour, and (c) adjust its actions accordingly when mediating the therapy intervention. Consequently, the robot software has three main sub-systems, one for sensing and interpretation of perceptual data, one for analysis of child behaviour, and a third for controlling the behaviour of the robot. The graphic user interface constitutes a fourth component.

One of our tasks in DREAM is to realize the robot software by integrating the individual contributions of five geographically-distributed teams of developers. The goal of this paper is to show how this task is facilitated by the adoption of CBSE, in general, and CPC, in particular. In the following, we summarize the main features of CBSE and CPC, and identify the associated software engineering practices that make this a suitable approach for the design and implementation of the DREAM robot software system. We then follow this with a description of our experience in putting these into effect in the implementation of the DREAM software architecture using placeholder components, focussing in particular on the merits this has for subsequent integration of the functional elements of the architecture.

3 Component-based Software Engineering

CBSE targets the development of reusable software [18] and has been widely adopted in the robotics community [19]. It complements traditional object-oriented programming by focussing on run-time composition of software rather than link-time composition. Consequently, it allows different programming languages, operating systems, and possibly communication middleware to be used in a given application. Related to the classic concept of communicating sequential processes (CSP) [20], components are individually-instantiated processes that communicate with each other by message-passing. Typically, component models assume asynchronous message-passing where the communicating component is non-blocking, whereas CSP assumed synchronous communication. The key idea is that components can act as reusable building blocks and that applications and system architectures can be designed by *composing* components. This gives rise to the two key concerns of component-based models: *composability* (the property of a component to be easily integrated into a larger system, i.e. to be reused under composition) and *compositionality* (the property of a system to exhibit predictable

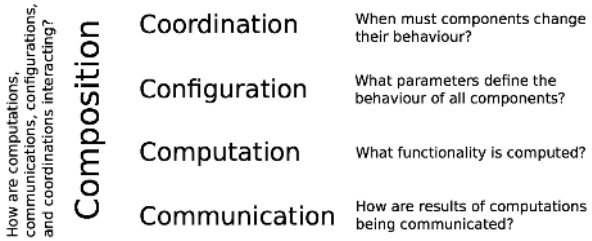

Fig. 1. The 5 Cs of the BRICS component model: the core idea is that good component-based engineering practice strives to decouple the design concerns of these five functions (from [21]).

performance and behaviour if the performance and behaviour of the components are known) [21].

In complex robotics systems, as in other large software systems, integration is difficult. It is made easier by adopting practices that support composability and compositionality. As Bruyninckx notes, composability and compositionality are not independent and "achieving full compositionality and composability is an ideal that is probably impossible to achieve" [22] because the information hiding characteristic inherent in good component design conflicts with the transparency required by system builders to optimize system robustness by selecting components whose internal design is such that it can cope with as many system variations as possible.

Although they share some common principles, CBSE and object-oriented approaches are not identical. Typically, the granularity of components in CBSE is larger than that of objects in object-oriented approaches. Thus, the functionality encapsulated in a component is usually greater than that of an object. Also, components are explicitly intended to be stand-alone independently-executable reusable pieces of software with well-defined public interfaces.

On the other hand, in CBSE, as in object-oriented programming, the component specification is separated from the component implementation. A component exposes its functionality through abstract interfaces that hide the underlying implementation. Thus, the implementation can be altered without affecting any of the systems or other components that use (i.e. interface with) that component. Components exchange data through their interface.

In robotics, a component implements a well-encapulated element of robot functionality and robot software systems are constructed by connecting components [6], [7]. The connections are often, but not necessarily made using ports in the components. As we will see below, this echoes the key elements of the CPC model.

4 CPC and The BRICS Component Model

Focussing on the issues that are particularly important in developing robot software, the BRICS Component Model (BCM) [21], [23], [24] is a best-practice CBSE model that provides a set of principles, guidelines, meta-models, and tools for structuring the development of individual components and component-based systems. It does so in a framework-nonspecific manner, i.e. at an abstract (meta) level that does not refer explicitly to the implementation of the required functionality on a specific robot platform.

BCM brings together two strands of software engineering: the separation of the so-called 4Cs of communication, computation, configuration and coordination [25] and robotics-oriented MDE [26]. BCM extends the 4Cs to 5Cs [3] by splitting configuration into finer-grained concepts of configuration and composition (see Fig. 1). The core principle is that good component-based engineering practice strives to decouple the design and implementation of these five concerns.

Computation refers to the system's main functionality, i.e. the information processing the component has been designed to perform. *Communication* provides the data required by the computation and takes care of the quality of service of data communication, e.g. timing, bandwith, loss (accuracy), priority, and latency. *Coordination* refers to the functionality that determines how all the components in a system work together and, thus, how the component or system behaves. *Configuration* refers to the functional aspects concerned with influencing the behaviour of the computation and communication elements either at start-up or at run-time. Finally, *composition*, the fifth C introduced in [21], provides the glue that binds together the other four Cs, each of which is focussed on decoupling functionality. In contrast, composition is very much concerned with coupling: how the other four Cs interact. Strictly, this C does not reflect software functionality but is rather a design characteristic that seeks to find a good trade-off between composability and compositionality [3].

Complementing CBSE, MDE aims to improve the process of code generation from abstract models that describe a domain. The OMG [8] defines four levels of model abstraction, going from higher to lower levels of domain specificity (i.e. from platform-independent to platform-specific) by adding platform knowledge. These four levels can be characterized as follows [21].

- M3 Domain-nonspecific: the highest level of abstraction using a meta-meta-model.
- M2 Platform-independent representation of a domain, using, e.g., a *Component-Port-Connector* (CPC) meta-model.
- M1 Platform-specific model: a concrete model of a specific robotic system but without using a specific programming language.
- M0 The implementation level of a specific robotic system, with defined software frameworks, libraries, and programming languages.

Level M2 is of particular interest here because the abstract model, i.e. the CPC meta-model, maps directly to the principles of component-based software engineering. The essence

of this model is as follows [21]. An application system has zero or more components and zero or more connectors. A component has zero or more ports, and a port belongs to one and only one component. A connector is always between two ports. Finally, and of particular importance in the context of the current discussion, components expose their data and service interfaces on their ports and exchange data over the attached connectors.

The principles of CBSE have another important role in that they also constrain the selection of a software environment that will be used to provide the abstraction layer between the application code, on the one hand, and the middleware and operating system services (including support of distributed systems processing and device input/output), on the other. This abstraction layer is typically provided by the robot programming framework. It provides an abstract interface between the robot software system (implemented as a network of components) and the underlying operating system and middleware.

The CBSE CPC approach offers several advantages in configuring robotics applications. The robot application can be implemented simply by identifying the components to be instantiated and specifying the connections between their port interfaces. Each component can encapsulate some coarse-grained robot functionality, i.e. the component can perform some substantial function while maintaining its cohesion. It is possible to have multiple instances of the same component by having a mechanism for uniquely naming each instance of the component and propagating that name to the component's port names. A key characteristic is the support for external configuration of the component allowing the behaviour of individual instances of a component to be customized by the application developer without recourse to the component developer. This goes hand in hand with a facility to allow components to run in different contexts, meaning that the robot programming framework allows the component user to specify the location of the components configuration files and other resources. Runtime configuration of the behaviour of the component can be provided to allow users and other components to control the component's processing, if required. In many robot programming frameworks, e.g. YARP, components run asynchronously on a distributed system with the inter-component communication over ports being handled transparently by the robot programming framework. This means that the logical definition of a component-based application is independent from its run-time configuration and the ports provide abstract component interfaces. This communication infrastructure may provide multiple transport layer protocols, e.g. udp, tcp, and mcast.

What is significant about this is that adopting the CBSE CPC approach imposes a minimal number of required configuration practices on the component developer. Specifically, these are to support:

1) Unique naming of each instance of a component and propagation of that name to the port names;
2) Renaming any of the ports through which the interfaces are exposed;
3) External configuration of the component behaviour;
4) External specification of the context (i.e. path) to search for the associated configuration files;
5) Run-time configuration of the component behaviour reading parameter values from a port.

Clearly, the underlying robot programming framework is pivotal in supporting these practices and making them straightforward for the component developer to implement. We consider this issue in the next section.

5 Robot Programming Framework

There are many robot programming frameworks. These include ROS [27], YARP [9], URBI [28], and Orca [29], [30]. All have their respective advantages and disadvantages, as explained in various surveys and comparisons [31], [32]. ROS, a popular choice in many instances, has been criticized in the context of CBSE as lacking an abstract component model [26]. On the other hand, while YARP does not have an abstract component model either, it provides a framework that explicitly supports all the CBSE needs identified above. We provide examples of the manner in which it does so in the following. Note that our adoption of YARP does not mean it is the only framework that can support the approach described in this paper; other frameworks, including ROS, would also be viable.

YARP was designed to support the CPC meta-model as well as the principles of CBSE and the principles of model-driven engineering. A YARP module corresponds to a component and it can have an arbitrary number of ports, with modules being connected by specifying the ports on either side of a connection. YARP supports the implementation of robot applications by identifying the components to be instantiated and specifying the connections between their port interfaces. As we will see below, YARP also provides considerable support for component development.

YARP supports coordination and configuration with the `RFModule` and `ResourceFinder` classes. It supports external configuration of the behaviour of the component through the use of configuration files. This allows the behaviour of individual instances of a component to be customized by the application developer or system integrator without recourse to the component developer. It is the responsibility of the component developer to expose all these parameters to the user. YARP provides the component developer with the `ResourceFinder` class that provides methods to read and parse key-value parameters either from the configuration file or from the arguments associated with a command-line invocation of that module. If key-value arguments are supplied in the configuration file and in the command-line invocation, the latter takes precedence. YARP supports coordination, i.e.

external control of component behaviour, through runtime configuration, by allowing commands to be issued on a special port with the same name as the component and by providing a dedicated `RFModule` method `respond` to handled input. These commands will typically alter the parameter values while the component is executing. The port can be used by other components and also interactively by a user through the YARP `rpc` directive. Parsing these parameter update commands is also handled by a utility method in the `ResourceFinder` class.

The functional aspect of the code — computation — is separated by localizing all functionality in a method derived from either the `Thread` class or the `RateThread` classes.

All communication with other components is effected through YARP ports. These are opened and closed during the configuration phase before the computation thread is instantiated and are passed as arguments to the thread.

YARP also provides a facility to allow components to run in different contexts. This means that it allows the component user to specify the location of the component's configuration files and other resources.

All YARP modules run and communicate asynchronously and YARP supports distributed computing allowing an application developer as well as a component developer to assign a component to a given run-server. It is straightforward to run components on a network of computers, be it a distributed system, a collection of PCs, or a server farm. It supports a flexible communication infrastructure with multiple transport layer protocols so that the ports can be configured to use different communication protocols and carriers, such as tcp, udp, mcast (multi-cast) with several variants of each.

6 DREAM System Architecture

Having laid the foundations with the CPC meta-model and the YARP robot programming framework, we are now in a position to describe the DREAM system architecture and its realization as an operational system that facilitates managed incremental integration of functional software.

System architectures in general, and cognitive architectures for robots in particular, are used as blueprints for the functional, structural, and behavioural specification of a complete system [33], [34], [35]. Typically, this specification is effected at quite a high level of abstraction and not at a level that is sufficient to address all the design decisions required before embarking on an implementation of the requisite functionality. However, because we are using CBSE in the form of the CPC meta-model, our specification of the DREAM system architecture can be rendered in terms that do allow it to be mapped directly to an implementation. This is accomplished by identifying the functionality of each major sub-system in terms of a suite of abstract functional primitives and the ports through which these primitives expose their abstract interfaces. Each sub-system can then be viewed as a macroscopic component with a well-defined set of ports. The system architecture then is completed by specifying the connectors, i.e. the interconnections between the ports associated with each sub-system component.

It is important to note here that these primitives represent a system specification that is grounded in user requirements derived from detailed use cases. This ensures that the inter-sub-system interfaces capture the semantics of the information exchanged and not just the syntax of the exchange protocol. The use cases define the interventions to be carried out between child, therapist, and robot. These were generated by developing a series of child-robot interaction scripts describing detailed scenarios for the three types of intervention being addressed: (a) joint attention, (b) imitation, and (c) turn-taking. The interventions were decomposed into nine tasks. For each task, we set out a detailed breakdown of the associated actions, component movements and sensory cues, and the associated sensory-motor processes. The sensory-motor processes provide the essential input for the definition of robot behaviour specification, i.e. the associated robot behaviours that have to be implemented to conduct all the intervention tasks, and the child behaviour specification, i.e. the expected child actions that have to be monitored. The robot behaviours are expressed in terms of six parameterized action primitives, e.g. `moveHead (x, y, z)`, and one vocal expression primitive. The child's actions are described in perceptual terms, from the perspective of the robot, as perception primitives that can be used to determine the child's behaviour, e.g. `identifyFaceExpression(x, y, z, expression_id)` or `getEyeGaze(x, y, z)`; see Table 1. While the translation from use case to primitive did not exploit a formal model of requirement elicitation, the resulting primitives, and their encapsulation in the port interfaces, ensure that the component-port-component communication in the operational system architecture embrace both the syntax and the semantics of the information exchange.

The implementation of the system architecture then becomes an exercise in developing and validating a placeholder component for each sub-system in the architecture, with each sub-system component emulating its constituent functional primitives and reading/writing the related data from/to the relevant ports. Thus, the complete system architecture is implemented first as a set of placeholder components comprising stubs for all perception and action primitives in each sub-system together with a complete implementation of the full set of port interfaces used by these stub primitives. Functionality is added as individual components are progressively integrated by allowing them to take over communication on the ports used by the stubs.

By adapting this operational-architecture implementation, we strike a balance between allowing the eventual builder of the individual primitive components the freedom to focus on algorithms and component functionality, on the one hand, and providing the necessary constraints on component interfaces

to ensure successful integration, i.e. compositionality, on the other. Each of these individual components must adhere to the strictly-defined interface semantics encapsulated in the placeholder port specification and implementation, exactly as advocated by Schlegel [36]. While we have not explicitly used Schlegel's set of five generic communication patterns, they are implicitly embraced by the YARP port communication protocols that implement the component interfaces.

Validation of the system architecture takes the form of ensuring that all the port interfaces and connectors are operating correctly. Since the number of ports and connectors can be large (127 and 76, respectively, in the case of the DREAM system architecture) this validation exercise can be non-trivial. This is because, with all components communicating asynchronously on all ports simultanously, it is difficult to monitor the behaviour of sender and receiver components and verify that communications have been sent and received correctly. We return to this issue later when we describe an approach based on discrete event simulation to model the invocation of each primitive stub in a controlled manner.

6.1 Architecture Overview

The core of the DREAM RET robot is its cognitive model which interprets sensory data (e.g. body movement and emotion appearance cues), uses these percepts to assess the child's behaviour by learning to map them to therapist-specified behavioural classes, and learns to map these child behaviours to appropriate therapist-specified robot actions. Thus, the DREAM system architecture has three major functional subsystems:

1) Sensing and Interpretation,
2) Child Behaviour Analysis, and
3) Robot Behaviour.

These sub-systems are implemented by three components, *sensoryInterpretation*, *childBehaviourClassification*, and *cognitiveControl*, respectively. The functional specifications of these three sub-systems are derived directly from the three types of intervention mentioned above (imitation, joint attention, and turn taking). These interventions are described as a sequence of actions, each action comprising a number of constituent movements and sensory cues linked to a particular sensory-motor process. The motor aspect of these processes provides the basis for the robot behaviour specification to be implemented in the cognitive control sub-system. The sensory aspect provides the basis for the sensory interpretation subsystems and also the child behaviour classification sub-system.

A walk-through of these interventions yielded a set of twenty-five sensory interpretation primitives, three child classification primitives, and seven cognitive control primitives. Each primitive will eventually be implemented as a distinct component in the DREAM architecture. However, for the purpose of implementation and validation of the system architecture, it is sufficient to group them together in the three sub-system placeholder components. These primitives are listed in Table 1. The input and output ports that are exposed by each primitive component and through which input and output data can be exchanged with these sub-systems (and, eventually, with the individual components that will encapsulate each primitive and replace the sub-system stubs) are also listed in Fig. 1. The parameters of each primitive in the three sub-systems are exposed by two dedicated ports, one for input and one for output, with the arguments encapsulated in a YARP vector or bottle,[3] whichever is more appropriate. The general naming convention for the two ports is to use the primitive name as the root of the port name, appending `:i` or `:o`, depending on whether it is an input or an output port, i.e. `/<primitive name>:i` for input and `/<primitive name>:o` for output. It only remains then to specify the connectors between the ports to define the sub-

Primitive	Input Parameters	Output Parameters
	sensoryInterpretation	
checkMutualGaze		returns true or false
getArmAngle		left_h,v,right_h,v
getBody		x, y, z
getBodyPose		<joint_i>
getEyeGaze	eye	x,y,z
getEyes		left_x,y,z,right_x,y,z
getFaces		<x,y,z>
getGripLocation	object_x,y,z	grip_x,y,z
getHands		<x,y,z>
getHead		x,y,z
getHeadGaze		x,y,z
getHeadGaze	x1,y1,z1,x2,y2,z2,x3,y3,z3	x,y,z
getObjects		<x,y,z>
getObjects	centre_x,y,z,radius	<x,y,z>
getObjectTableDistance	object_x,y,z	vertical_distance
getSoundDirection	threshold	azimuth, elevation
identifyFace	x,y,z	face_id
identifyFaceExpression	x,y,z	expression_id
identifyObject	x,y,z	object_id
identifyTrajectory	<x,y,z,t>	trajectory_descriptor
identifyVoice		voice_descriptor
recognizeSpeech		text
trackFace	seed_x,y,z,time_interval	projected_x,y,z
trackHand	seed_x,y,z,time_interval	projected_x,y,z
trackObject	seed_x,y,z,time_interval	projected_x,y,z
	childBehaviourClassification	
getChildBehaviour		<state, probability>
getChildMotivation		engagement, confidence
getChildPerformance		performance, confidence
	cognitiveControl	
grip	state	
moveHand	handDescriptor,x,y,z,roll	
moveHead	x,y,z	
moveSequence	sequenceDescriptor	
moveTorso	x,y,z	
say	text, tone	
enableRobot	state	
getInterventionState		id, state, cogMode

TABLE 1
The primitives provided by the three sub-system placeholder components. The input and output parameters are exposed through the matching ports, e.g. `/sensoryInterpretation/getEyeGaze:i` and `/sensoryInterpretation/getEyeGaze:o`.

3. YARP has a variety of methods to support abstraction in data representation, including the Bottle class. This is a simple collection of objects that can be described and transmitted in a portable way. This class has a well-defined, documented representation in both binary and text form. Objects are stored in a list, which you can add to and access.

system interconnectivity and complete the specification of the DREAM system architecture, which we will do in the following sections. First we will expand on the functional specification of the three sub-system placeholder components.

6.2 The *sensoryInterpretation* Component

The functionality of *sensoryInterpretation* is specified by the twenty-five perception primitives and associated input and output ports in Table 1. Not all primitives have input parameters. The components for those that do are stateful, i.e. once the associated argument values are set, they remain persistently in that state until reset by another input.

6.3 The *childBehaviourClassification* Component

The `getChildBehaviour()` primitive classifies the child's behaviour on the basis of current percepts. It produces a set of number pairs where the first element of each pair represents a child state and the second element the likelihood that the child is in that state. Thus, the primitive effectively produces a discrete probability distribution across the space of child states.

The `getChildMotivation()` primitive determines the degree of motivation and engagement on the basis of the temporal sequence of child behaviour states, quantifying the extent the children are motivated to participate in the tasks with the robot and detect in particular when their attention is lost. It produces two numbers, the first representing an estimate of the degree of engagement and the second representing an indication of confidence in that estimate.

The `getChildPerformance()` primitive determines the degree of performance of the child on the basis of a temporal sequence of child behaviour states, quantifying the performance of the children in the therapeutic sessions. It produces two numbers, the first representing an estimate of the degree of performance and the second representing an indication of confidence in that estimate.

6.4 The *cognitiveControl* Component

The functionality of the *cognitiveControl* component is specified partially by the first seven action primitives listed in Table 1. This is only a partial specification because the basis for invoking each of these action primitives cannot be fully defined (whereas, in the case of *sensoryInterpretation*, all of the primitives are continually invoked to monitor the status of the robot's environment). Specifically, these seven action primitives only reflect the functionality needed to carry out the robot behaviours to follow the scripted interventions that are encapsulated in the use cases. However, by itself this is not sufficient because there are two additional considerations for effective robot control in the context of supervised or shared autonomy.

First, the robot control is focussed on interaction with the child. Even when everything goes according to plan and the intervention scripts can be followed exactly, there is a need to adapt the robot behaviour to ensure its actions are synchronized with those of the child. In a sense, the robot's behaviour, in an ideal situation, is entrained by the child's (or, to be more exact, the child's and the robot's actions are mutually entrained). This means that the robot control is not just a simple case of effecting playback of scripted motions and there has to be an element of adaptivity in the robot behaviour, even in the case of following the scripted interventions.

Second, individual children are likely to deviate from the generalized expectations underlying the scripted interventions and the stereotyped standard behaviours they describe. Even slight deviations call for adaptivity on the part of the robot. The *childBehaviourClassification* sub-system provides crucial information on the extent of these deviations in the guise of the engagement and performance indicators. As these deviate from acceptable norms, the robot controller has to adapt, either to suspend the interaction and hand over control of the situation to the therapist, or to select some action — autonomously — that will re-engage the child and hopefully improve his or her performance. This adaptivity requires more cognitive control. The component exposes the current status of its control behaviour so that this can be used by the other two placeholder components. Specifically, the *cognitiveControl* placeholder component has a `getInterventionState()` primitive and an associated output port that identifies the intervention that is currently being enacted and the current phase within that intervention. To facilitate this, each intervention is defined as a finite state automaton (equivalently, a state transition diagram) with uniquely-labelled states to provide a common reference for all other components.

While a `getInterventionState()` primitive handles the situation where the robot control is following the prescribed intervention, even when the robot is adaptively entrained with the child's behaviour, it cannot capture the cognitive behaviour of the robot when dealing with situations that deviate from this script because of diminished performance or engagement on the part of the child. This is true because cognition and autonomy — even supervised autonomy — by definition precludes complete *a priori* prescriptive control of an agent's behaviour by an external agent, including a software designer: the agent in question (in this case, the robot) decides its own behaviour. To compensate for this, the `getInterventionState()` primitive has a third output parameter to flag situations when the controller is (cognitively) handling an unexpected deviation from a given intervention state (typically as a result of the child's behaviour exhibiting diminished engagement and/or performance in the current intervention). This allows the other components, and the user (either a therapist or a software developer), to be informed about what is going on in the cognitive controller, at least in general terms.

In summary, in contrast to the situation with the other two

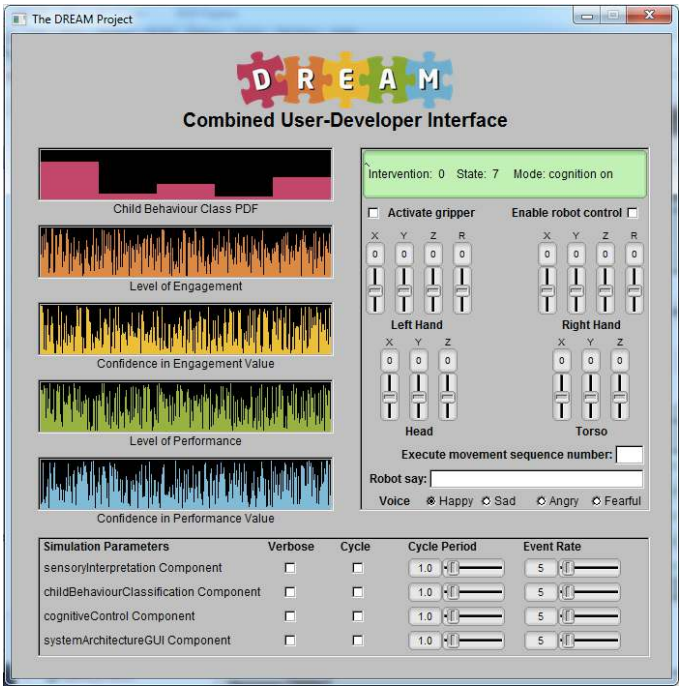

Fig. 2. The system architecture graphic user interface.

placeholder components, the primitives alone are inherently insufficient to fully specify the adaptive behaviour facilitated by cognitive control for the reasons set out above. Thus, the functionality for adaptation is not specified by the primitives and the associated parameters. From the perspective of the system architecture, this is not problematic provided the information exposed by the other components, and accessed through the ports specified in this component, are sufficient. If it turns out they are not, then we will have to extend the system architecture specification and implement new primitives and associated ports. To an extent, this demonstrates the power and flexibility of the operational-architecture approach described in the paper in so far as it shows the advantages of balancing the imposition off rigorous inter-component specification on developers with allowing them freedom to implement the internal functionality, something we stress in the introduction.

6.5 The *systemArchitectureGUI* Component

In addition to the three sub-system placeholder components identified above, there is one other utility component in the system architecture. This is a Graphic User Interface (GUI) to facilitate external control of the robot by a user (either a therapist or a software developer) and to provide the user with an easily-to-understand view on the current status of the robot control. It also provides a graphic rendering of the child's behavioural state, degree of engagement, and degree of performance in the current intervention. All GUI functionality is accomplished by reading and writing data to ports connected to the three placeholder components, i.e. using the primitive interfaces listed in Fig. 1. Essentially, the GUI re-presents the information that is made available on the output ports of the *childBehaviourClassification* and the *cognitiveControl* components and allows the user to exercise the *cognitiveControl* component's input ports. It also allows the system integrator to set the parameters associated with the validation of the system architecture through discrete event simulation (see Section 8.3).

A screen-shot of the GUI is shown in Fig. 2. There are three main areas in the GUI.

The top left-hand area comprises five displays that visualize the child behaviour classification probability density function (PDF), the child's level of engagement and a measure of confidence in that value, as well as the child's level of performance and a measure of confidence in that value. As this screenshot was acquired during a test of the system architecture using the sub-system placeholder components, the data being visualized in the screenshot is purely random. When the random data produced by the discrete event simulation is replaced by valid data after the *childBehaviourClassification* component functionality is implemented, the information displayed will be more meaningful. Of course, this requires no change in the implementation of the GUI since it already adheres to the architecture specifications.

The top right-hand region comprises a display to show the current status of the *cognitiveControl* component (i.e. the output of the `getInterventionState()` primitive) and a suite of interface widgets to control the robot using the other *cognitiveControl* component primitives.

Finally, the bottom of the GUI comprises a suite of widgets to set various options in the operation of each of the three sub-system placeholder components: whether or not to operate in verbose mode (echoing input and output messages to the terminal), whether or not to cycle through the invocation of each primitive in each component one at a time to facilitate validation of port connections, the cycle period to be used, and the number of primitive invocations to simulate every second. These operational parameters are explained in more detail in Section 8.3 below when we discuss the use of discrete event simulation to aid the validation of the system architecture.

6.6 Inter-connectivity between the Components

Any component that needs to access the information exposed on the ports associated with a primitive has to have equivalent ports of its own (so that the two ports can be connected) but reversing the input/output designation. Thus, for example, for the `identifyObject(x,y,z,object_id)` primitive one would connect `/cognitiveControl/identifyObject:o` to `/sensoryInterpretation/identifyObject:i` and `/sensoryInterpretation/identifyObject:o` to `/cognitiveControl/identifyObject:i`. This allows `cognitiveControl` to send the x, y, and z location of the

object to be identified to `sensoryInterpretation` and then to receive the identification number of that object from `sensoryInterpretation`.

Regarding the connectivity between the four components in the DREAM system architecture, the following principles apply.

Each *sensoryInterpretation* output port is connected to the counterpart input port in the *cognitiveControl* and *childBehaviourClassification* components.

Each *sensoryInterpretation* input port is connected to the counterpart output port in the *cognitiveControl* component (but not the *childBehaviourClassification* component).

Each *childBehaviourClassification* output port is connected to the counterpart input port in the *cognitiveControl* component.

All of the *cognitiveControl* input and output ports will be connected to the GUI component *systemArchitectureGUI* to facilitate external control of the robot by a user (either a therapist or a software developer) and to provide the user with an easily-comprehended view on the current status of the robot control. Furthermore, the child state, degree of engagement, and degree of performance output ports in the *childBehaviourClassification* component will also be connected to the GUI and graphically rendered in a suitable manner.

7 Software Engineering Standards and Integration Procedure

As we noted in the introduction, an operational implemented architecture as a driver for sub-system development and integration is insufficient by itself: you also need the user requirements and system specification that define the software functionality as well as an effective software engineering process. We addressed the user requirements in the previous section and here we address briefly the software engineering standards that underpin the development process, focussing in particular on the quality assurance process in which the integration procedure is embedded. Our strategy was to adopt a set of software engineering standards for each phase of the software development life-cycle — from component and sub-system specification, through component design, component implementation, and component and sub-system testing, to documentation — but to keep these as minimal as possible while at the same time not compromising software quality, placing special emphasis on the latter phases of the life-cycle to facilitate effective system integration.

We also adopted a set of procedures for quality assurance, focussing in particular on the procedures by which software is submitted for integration, tested, and checked against the software engineering standards.

We distinguish between *recommended standards* that reflect desirable practices and *mandatory standards* that reflect required practices. DREAM component software developers and robot application developers are strongly encouraged to adhere to the desirable practices but these standards do not form part of the criteria that will be used to decide whether or not a given component of application can be included in the DREAM software repository: that is, they do not form part of the DREAM software quality assurance process governing component integration. On the other hand, the required practices *do* form part of the software quality assurance process and a component or application will only be accepted for integration in the release version if it complies with the corresponding mandatory standards.

In creating these standards, we have drawn from several sources, including the GNU Coding Standards [37], Java Code Conventions [38], C++ Coding Standard [39], and the Doxygen User Manual [40].

7.1 Component Specification

Our operational-architecture approach aims to allow researchers and software developers as much freedom as possible in the specification of the components that meet the functional requirements outlined above. Consequently, this phase of the software development life-cycle are subject to *recommended* standards stipulating the required practices for requirements eliciation, documentation of the underlying computational model, the functional specification (including a class and class-hierarchy definition), the data model (summarized by an entity-relationship diagram, data dictionary, input functional, control, system configuration data, output functional, control, system configuration data for each process or thread in the component, and an object-relationship model where appropriate), a process flow model (summarized using a data-flow diagram (DFD), identifying data flow, control flow, and persistent data sources and sinks), and a behavioural model (summarized with a state transition diagram and an object-behaviour model, where appropriate).

7.2 Component Design

The principles of good design have already been outlined the guidelines set out in Section 4. We consider these guidelines to be a set of *recommended standards* for component design and we will not repeat them here. However, it is important to note that these guidelines give rise to several essential practices — and mandatory standards — in component implementation, as follows.

7.3 Component Implementation

The mandatory standards for implementation of components are the first set of standards that form part of the software quality assurance process and the component or application will only be accepted for integration in the release version if it complies with these standards.

The mandatory implementation standards address file names and file organization, internal source code documentation,

and component functionality, targeting the 4Cs of configuration, coordination, computation, and communication. As noted above, these mainly focus on the three Cs of configuration, coordination, and communication because these are the ones that impact most on integration. Mandatory standards for the fourth C, computation, only relate to the manner in which the functionality is encapsulated using YARP (see Footnote 2).

The recommended implementation standards provide extensive guidelines on programming style and programming practice, guidelines which would typically be mandatory in industry but are very hard to enforce in collaborative research projects (see [41] for details).

All standards, mandatory and recommended, make reference to an implementation of a prototype component which is available to all developers.

7.4 Testing

DREAM software is subject to a spectrum of test procedures, including black-box unit tests, white-box structural tests, regression tests, and acceptance tests.

Black-box testing is a testing strategy that checks the behaviour of a software unit — in this case a component — without being concerned with what is going on inside the unit'. Typically, it does this by providing a representative sample of input data (both valid and invalid), by describing the expected results, and then by running the component against this test data to see whether or not the expected results are achieved. Component software developers must provide a unit test with every component submitted for integration into the DREAM release and this unit test must be compliant with several requirements concerning application launch, data sources and data sinks, filenames, file location, and test instructions. These instructions should explain how the four Cs of communication, configuration, computation, and coordination are exercised by the test.

White-box testing is a testing strategy that checks the behaviour of a software unit or collection of units in a (sub-) system by exercising all the possible execution paths within that unit or system. Thus, white-box testing differes fundamentally from black-box testing in that it is explicitly concerned with what is going on inside the unit. In DREAM we perform white-box testing on a system-level only. Components are not subject to white-box testing when being integrated into the DREAM software release, although developers themselves are encouraged to use white-box testing before submitting the component for integration.

Regression testing refers to the practice of re-running all integration tests — black-box and white-box — periodically to ensure that no unintentional changes have been introduced during the ongoing development of the DREAM software release. These tests check for backward compatibility, ensuring that what used to work in the past remains working.

DREAM software is also subject to periodic qualitative assessment by psychotherapist practitioners to validate the behaviour and performance of the system against their requirements. These tests take place whenever a new version of the DREAM software release is made available to the practitioners.

7.5 Quality Assurance Process and Integration Procedure

Software that is to be included in the release branch of the DREAM software repository must comply with the mandatory standards outlines above. A strict procedure is followed to validate and test software developed by the partners prior to integration into the release version. This takes the form of a 57-point check-list organized under four headings derived from the mandatory standards: Files and Directories, Internal Source Code Documentation, Component Functionality, and Component Unit Testing (see [42] for details).

8 Validation

We noted in the introduction that the contribution of the paper is primarily methodological, with secondary support for that methodology being provided by the discrete event simulation technique to facilitate handling the complexity of integration that naturally arises from a complete implementation of the full system architecture and the large number of associated inter-sub-system port communications. We note also that this is a relatively straightforward approach that provides clear integration acceptance criteria, i.e. hard constraints on the software sub-system development outcome, without imposing hard constraints on all the other phases of the software development lifecycle, thereby substantially easing the burden on the developers without compromising the integration process.

Thus, there are two aspects to be validated: (1) the impact this methodology has had on the developers' software engineering practices, and (2) the impact it has had on the integration process. These two issues are addressed in the following sections, the first by reporting an impact analysis on our team of developers and we address the second by describing our experience in integrating software that has been submitted by developers for integration. We also address a third element of validation concerning the authentication of the inter-component communications between the sub-systems in the architecture. Since the number of communication ports can be large, we propose in Section 8.3 the use of a discrete event simulation technique to facilitate handling the complexity that naturally arises from a complete implementation of the fully operational system architecture.

8.1 Assessment of Impact on Developers' Software Engineering Practices

The success — or failure — of the operational-architecture component-based software engineering approach to system integration being described and advocated in this paper hinges

of three things: (a) the architecture needs to successfully encapsulate the user requirements, (b) the information exchanges exposed by the architecture component sub-system interfaces must be sufficient for the sub-system components to satisfy these requirements when integrated, and (c) the developers have to comply with the software engineering standards and quality assurance process on which successful integration depends. Furthermore, to be truly effective, the developers need to believe that the time and effort to achieve compliance is worthwhile: that the process is effective and efficient, that the right balance of freedom and constraint has been struck, and that the operational system architecture facilitates their work. As we noted at the outset, system software integration is a socio-technical problem, requiring both appropriate tools and acceptance of the value of the associated methodologies by the developers. In this section, we illustrate this acceptance in the DREAM developer team in two ways.

First, we report on a poll of developers' views taken approximately one year after the launch of the standards, quality assurance process, and the operational system architecture components and example application. This poll was taken to determine what needs to be changed in the software standards, integration procedure, system architecture, and support documentation and software, and to gauge the impact of the operational system architecture on their engineering practices in the context of the DREAM project.

The poll took the form of a multiple choice questionnaire with a small number of questions (12) formulated in a manner that elicited feedback, positive or negative, as well as inviting comments on what needed to be removed, changed, or added for each question topic. The initial version of the questionnaire had more questions but we reduced the number to increase the likelihood of obtaining meaningful responses. Each question offered five choices: Strongly agree; Agree; Neither agree nor disagree (I don't know / I don't have an opinion); Disagree; and Strongly disagree. Questions were formulated both positively and negatively so that some questions required agreement to express a positive response with other questions required disagreement to express a positive response. In this way, we attempted to eliminate biased responses as a consequence of the manner in which the questions were framed. Furthermore, views on each issue (e.g. the usefulness of the system architecture) were elicited using more than one question; conclusions are drawn only if the answers are consistent. The questionnaire was completed by each team of developers in the consortium.[4] This resulted in four sets of responses. Clearly, the results do not have the statistical significance that would derive from a canvass of a much wider deployment of the approach we describe in the paper but they are indicative nonetheless of the perceived value of the approach by the team of developers involved in the DREAM research project.

4. The questionnaire and an anonymous version of the results are available at www.dream2020.eu/software_questionnaire.

The results of the poll were very positive: developers clearly felt that the balance of mandatory and recommended software engineering standards allows sufficient flexibility to design their own code and that the time and effort required to familiarize themselves with the mandatory standards was worthwhile. Significantly, they did not think there should be fewer mandatory standards and they agreed that the support software, i.e. the protoype component and example applications, helped illustrate the standards and assisted them in writing their own components. Crucially, there was a very strong consensus that the operational system architecture made it easier to design software that can be integrated into the DREAM RET application and they prefer this approach to a more standard paper-based description of the architecture. There was also agreement that the system architecture provided sufficient specification of how their software should interface to other subsystems. With one exception, they also felt that the working architecture with operational port interfaces makes it easier to design software that can be integrated (the operational port interfaces makes it harder in the sense of raising the threshold on correctness before a component can be submitted for integration). Developers were less impressed with the clarity of the integration procedure and this clearly needs to be streamlined. Of course it will only be possible to evaluate the long-term impact of the approach described in this paper towards or after the end of the project. However, the initial phase of the development effort is critical and if the approach does indeed facilitate effective development practices, manifested by efficient integration, then its impact will be felt not only near the end of the project after much training but also at the outset as it promotes the adoption of effective practices in the start-up phase. This is exactly what we have found so far in the project.

Second, it is noteworthy that the acceptance — and value — of the standards and the operational system architecture has been manifested in an unexpected manner. One of the developers produced, unforeseen and unsolicited, a YARP component generator utility that produces the essentials of a component with all the required port interfaces in a manner that is fully compliant with all the mandatory software standards for the 4Cs and, therefore, with most of the integration procedure (it does not take care of the unit test applications). While this does not validate the efficacy of the operational system architecture, *per se*, it does validate the acceptance of the process on which the architecture is founded.

8.2 Assessment of Impact on the Integration Process

While, as we noted above, it is still relatively early in the DREAM project, our experience with integration allows us to draw some important conclusions. First, the system architecture, viewed (as we said above) as a driver for both sub-system development and integration, has been the cause of a prolongation of the development period. That is not to say that it has

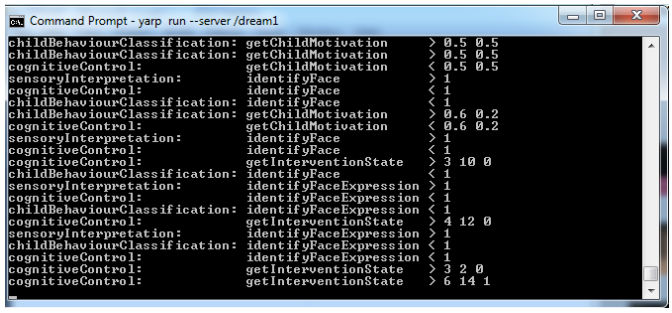

Fig. 3. The information directed to the terminal when the sub-system placeholder components are operating in verbose mode.

delayed development but that it has inhibited the submission of software for integration before it was really ready. This is of course a very positive outcome: it minimizes unecessary testing of immature software and maximizes effectiveness of the integration effort. There are two reasons for this. First, the effort required to create an operational system architecture, as opposed to a paper-based system architecture that could easily be updated, focussed the entire team's collective mind on producing a strong set of user requirements (based on use case analysis) and an attendant functional specification (encapsulated in the perception and action primitives by which each sub-system exposes its functionality). Second, the port interfaces that realize the primitive-based sub-system interaction form a set of explicit compatibility acceptance criteria — both syntactic and semantic, as we argued above — with which any new component must be consistent. These, together with the mandatory standards that address the 4Cs, provide clear and explicit targets which developers know will be scrutinized in the quality assurance process, and specificially in the integration procedure checklist. Again, this concentrates the collective mind of the developers and has so far at least tended to produce high-quality software by the time it is deemed ready for submission for integration.

8.3 Validation of the Operation of the System Architecture by Discrete Event Simulation

As we have seen, the DREAM system architecture comprises three main sub-systems: (1) Sensing and Interpretation, (2) Child Behaviour Analysis, and (3) Robot Behaviour. Initially, these three sub-systems are implemented by three corresponding placeholder components: *sensoryInterpretation*, *childBehaviourClassification*, and *cognitiveControl*. Each component provides stub functionality for the defining perception and action primitives listed in Table 1. Each placeholder component reads any input parameters from the input ports associated with its constituent primitives (where necessary) and writes simulated results (i.e. random values) to the associated output port. If verbose mode is enabled, each component echoes to the terminal the data that was read and written (see Fig. 3).

A placeholder component does nothing except send and receive simulated data on each of the ports defined in the sub-system specification. For the purposes of this validation, no cognizance is taken of stateful computation. The purpose of the placeholder is to provide a working integration framework that realizes all component interfaces and exposes all data produced by the various perception and action primitives on the component ports.[5] As we mention above, since the number of ports and connectors can be large, the validation of the interfaces can be difficult: there is often too much information to keep track of when matching data sent and received on the many ports used by asynchronously communicating components. One possible solution is to use an explicit coordinator system [43] and embed data flow port monitoring in the functional components [44]. Since we are more interested in facilitating the integration process rather than effecting robust coordination between components, an alternative solution is to control the rate at which messages are sent and received so that input and output messages can be matched and validated while at the same time ensuring that asynchronous and concurrent communication is still respected. This has been done by having each placeholder component operate as a real-time discrete event simulation in which events correspond to the execution of a perception, classification, or action primitive. An event occurs when a primitive returns a value. To make this simulation realistic, especially from the perspective of asynchronous concurrent communication, in our implementation the invocation of a given primitive follows a probabilistic model. We assume that the events in question, i.e. the primitive invocations, are random and are drawn from a Poisson probability distribution [45], with each sub-system having its own event rate.[6] Since the behaviour of the children that interact with the robot can sometimes be unpredictable, for technical and methodological reasons we assume for the purposes of validating the implementation of the system architecture that the primitives which convey information about their behaviour are invoked at random. Our working hypothesis is that if the system architecture can be shown to work with randomly occurring data then it is likely to work with data that flows more regularly. It is possible to set all but one event rates to zero for a certain period of time, and to cycle through each primitive as the simulation proceeds. This makes it easier to check the operation of the component

5. There are inherent limitiations in this form of port validation, such as ensuring that the information exchanged on the port satisfies some semantic condition. If one wants the certainty that the components operate as specified, one needs to adopt a more rigorous and formal methodology and software engineering practice.

6. If events occur at random in time then the number of events in a fixed interval of time has a Poisson distribution. That is, the probability that an event X takes on a value x (the probability that the number of times a primitive is invoked equals x) is given by the expression $P(X = x) = \frac{e^{-\lambda}\lambda^x}{x!}, x = 0, 1, 2, \ldots$ where λ is the event rate (the average number of times a primitive is invoked in that interval of time). Appendix 1 explains how the Poisson distribution is sampled.

Listing 1. Pseudo-code algorithm for discrete event simulation of placeholder component

```
while (isStopping() != true) {

   // process simulation parameters based on how much time has passed

   compute elapsedTime (since previous iteration)
   compute cumulativeTime (since current cycle started)

   if (cumulativeTime > cyclePeriod) {
      reset cumulativeTime

      if (cycle flag is set) {  // go through primitives one at a time
         set event rate for all primitives to zero, except the target primitive
         advance the target primitive for next cycle
      }
      else {                    // go through primitives simultaneously
         set event rate for all primitives  to the parameter value
      }
   }

   for all input ports that do not provide input arguments for primitives
   being simulated in placeholder component X {
      read input port in non-blocking mode
      if verbose mode
         write data read to terminal
   }

   for all primitives being simulated in placeholder component X {
      read associated input port in non-blocking mode (if it exists)
      if verbose mode
         write data read to terminal

      /* compute Poisson distribution mean and sample Poisson distribution      */
      /* to determine number of events (i.e. primitive invocations) to simulate */

      lambda = eventRate[current_primitive++] * elapsedTime;
      count = samplePoisson(lambda);

      for (i=0; i<count; i++) {
         generate random data
         write data to the output port associated with this primitive
         if verbose mode
            write data read to terminal
      }
   }
}
```

especially when all four placeholder components are working together.

There are two phases in each iteration of the discrete event loop. In the first phase, all input ports that do not provide input arguments for primitives being simulated in the placeholder component are read in non-blocking mode, again echoing data read to the terminal when in verbose mode. In the second phase, an invocation of each primitive in this placeholder component is simulated by reading the input port associated with that primitive in non-blocking mode (if it exists). If in verbose mode, this data is echoed to the terminal. Then, the Poisson distribution is sampled to determine how many times that primitive has been called in the time elapsed since the previous iteration of the discrete event simulation loop (for details of the sampling process, see Appendix 1). This determines how many times the primitives output should be written to the associated output port. Random data is generated for each invocation and written to the associated output port and, if in verbose mode, the data is also echoed to the terminal. Listing 1 gives a pseudo-code algorithm for this simulation process. Note that the port named after the component and used for coordination (see Section 5) is not exercised in this process since the command protocol is specified by the component functionality and not by the system architecture and the use-case primitives. Once specified it can be included.

Four simulation parameters are provided to allow the developer to control the simulation without having to recompile the placeholder code. These are:

verbose
: If this parameter has the value *on*, messages will be sent to the run server terminal to echo the input and output to/from each primitive; if *off* no messages are sent.

cycle
: If this parameter has the value *on*, only one primitive is set to have a non-zero event rate in each iteration of the discrete event simulation loop and therefore only one primitive will output data at the specified rate in any one iteration of the discrete event simulation loop. If the parameter has the value *off* all primitives have the specified non-zero event rate and all will output data at the rate determined by the probabilistic model.

cycle_period
: This is the time period for which a primitive is the only one to have a non-zero event rate.

event_rate
: This is the number of events (i.e. primitive invocations) per second. In the current implementation, the same event rate is applied to all primitives except in cycle mode. It would be more flexible to have individual rates for each primitive but we have not found this to be necessary.

This event rate determines the Poisson distribution parameter, i.e. the mean number of events, computed as the event rate multiplied by the time taken to execute one discrete event loop. This event loop time is measured in real-time and is not a fixed discrete event simulation interval. Again, this produces a more realistic simulation to validate communication over the port connector.

This discrete event simulation provides a useful framework for integration of new components. As we mentioned already, the functionality of each sub-system is being developed incrementally with new components being developed to implement each primitive. Integration of these new primitives is made easier by simply setting the event rate of the stub versions in this placeholder to zero, allowing the newly-integrated component to read and write the requisite data to the relevant ports instead of the placeholder sub-system component.

9 Conclusions

In this paper, we have argued that the adoption of CBSE in the guise of the CPC meta-model leads to a natural and effective method of system architecture design and implementation that facilitates orderly integration but without imposing excessively strong engineering standards which might not be adopted by every team of developers. This lightweight aspect is important as it is attractive for collaboratively-undertaken system development where, unlike some more stringent industrial

development environments, developers voluntarily agree to adhere to the software engineering practices. In concert with this, of course, is the need to support developers in adhering to these guidelines. We have shown that the YARP middleware is an effective way of doing this. While YARP was designed specifically for the robotics domain, and provides many useful robotics and image processing tools, it is inherently general-purpose and makes for a very effective thin support environment for any system that adopts the component-port-connector meta-model, irrespective of its application domain.

One of the principle advantages of the CPC meta-model is that it solves the problem of coarse-grained software reuse in a straightforward manner if the practice of implementing components so that they can be externally reconfigured is strictly enforced. There are five aspects to this reconfiguration. First, there is the ability to provide a unique name for each instance of a component and propagation of that name to the port names. Second, there is the ability to recast the component interface by renaming any of the ports through which the interfaces are exposed. Third, there is the ability to configure the component behaviour by reading all parameter values either from a configuration file or the command line invocation. Fourth, there is the ability to automatically reconfigure the paths where the component searches for component resources (e.g. configuration and data files) by modifying a specific external configuration file rather than by recompiling the component software. Fifth, there is the ability to alter the operational behaviour of the component either at invocation-time by setting component parameters through command line arguments or through configuration files, or at run-time through a standard component-specific port. Support for the developer by the underlying robot programming environment is crucial and YARP provides all the required mechanisms.

A second major advantage of this approach is that it provides for the direct instantiation of a system architecture by realizing sub-systems as placeholder components that provide stubs for each constituent primitive and by exposing all associated functionality through the component ports. Sub-system placeholder stub functionality is then incrementally replaced with the final functional components. This operational-architecture approach to system development prioritizes the role of system integration by focussing explicitly on the means by which components will be integrated, i.e. by instantiating the connectors between component ports, and — most importantly — by providing a working implementation of these sub-system interfaces through the sub-system placeholder components. This reduces the complexity of the integration process significantly and provides developers with a clear template for their component port interfaces.

Finally, we have shown how the difficult task of validating the interaction between components in a realistic large-scale system architecture with 76 connectors can be eased significantly by implementing the placeholder sub-system components as a discrete event simulation. This allows the system integrator to adjust the rate at which the constituent stub primitives are invoked and to control their invocation either by cycling through them, invoking them one at a time, or more realistically by allowing all of them to be invoked in any one discrete event simulation time quantum according to a probabilistic model following a Poisson distribution. By giving the developer access to the four controls that govern the discrete event simulation behaviour of each component (i.e. verbose mode echoing port data to the run-server terminal; rate of primitive invocation; cyclic invocation or concurrent invocation; and cycle time) the operation of the complete system can be validated more easily, as can the integration of a functional component by overriding input/output of the associated placeholder stub primitive, while still respecting the asynchronous and concurrent character of communication in CBSE.

Large-scale system development raises many problems, not least of which is the difficulty in managing the development and integration processes. Our experience in developing the DREAM system architecture has shown that the CBSE approach described here — the use of the component-port-connector meta-model, adherence to the five practices associated with designing components to facilitate external configuration, the use of the YARP middleware platform to provide functional support for these guidelines, and the use of discrete event simulation to control the behaviour of stub primitive input/output — greatly eases the management of both processes. While CBSE is often used for facilitating reusability, we have demonstrated here another benefit: substantially facilitating the development and integration process in collaborative, and often geographically-distributed, research projects. It strikes a good balance between software engineering best practice and the socio-technical problem of managing effective integration of software developed by researchers that are not always inclined to adopt extensive strict standards.

Before concluding, a final caveat is in order. In an ideal world, all developers would adhere to professional software engineering standards and state-of-the-art software engineering practices, especially with respect to formally- and semantically-defined functional and non-functional requirements and capabilities, e.g. model-driven engineering approaches. We are not suggesting that the approach described offers an alternative to these more sophisticated approaches. It is instead a compromise that adopts methodologically-sound practices focussed on three of the four Cs in CBSE with minimal overhead in effort. If one requires greater assurance of correctly specified and implemented software, then one has little choice but to make the necessary effort to adopt more sophisticated software engineering tools than those discussed in this article. Hopefully, all software, even software developed in collaborative research projects, will eventually be developed using sophisticated tool chains. In the meantime, CBSE and the operational-architecture approach described here provide a useful compromise.

APPENDIX 1 SIMULATION OF RANDOM EVENTS

The system architecture described in this paper is initially implemented by three sub-system placeholder components, each of which simulates the invocation of the suite of primitives that constitute that sub-system. To make this simulation realistic, the invocation of a given primitive follows a probabilistic model. We assume that the events in question, i.e. the primitive invocations, are random and are drawn from a Poisson probability distribution. The goal, then, is to provide a function that can sample a Poisson distribution to determine the number of invocations that occur in a given period of time [45].

For discrete event simulation, we divide time up into discrete intervals of finite duration, *e.g.* 5 seconds and simulate the behaviour of the system by computing what happens in each interval. For a simulation of primitive invocation, the question we need to answer is: 'How many times was that primitive invoked in that finite interval?' To answer this question, we need to know the probability that primitive being invoked in any given interval. Actually, we need more: we need to know the probability that the primitive being invoked once, twice, three times, and so on.

If events occur at random in time then the number of events in a fixed interval of time has a Poisson distribution. That is, the probability that an event X takes on a value x (the probability that number of cars arriving equals x) is given by the expression

$$P(X = x) = \frac{e^{-\lambda}\lambda^x}{x!} \qquad x = 0, 1, 2, \ldots \quad (1)$$

where λ is the arrival rate (the average number of cars in that interval of time). So, for example, if the invocation rate is two calls every second, then the probability of three invocations (in any five second interval) is given by

$$P(X = 3) = \frac{e^{-2}2^3}{3!} = 0.18 \quad (2)$$

In more formal terms, Equation 1 is a probability mass function of a discrete random variable X (*i.e.* X varies or occurs randomly) and $P(X = x)$ is the probability mass function which defines the probability that X takes on the value x. The probability mass function is written

$$f(x) = P(X = x) \quad (3)$$

The variable x is referred to as a random variable whereas X is a random sample of a probability distribution.

The random variable is usually but not always (as we will see later) a uniform random variable. This means that it takes on values in the interval $[0 - 1]$ at random, with uniform probability of each value occurring. This is typically what a random number generator tries to achieve.

In general, this allows us to work out the probability that, *e.g.*, a given primitive will be invoked n times in any one interval of time. *However, this is not what we want.* We require the number of times a primitive is *actually* invoked in any specified interval of time, given that the invocation rate has a Poisson distribution with (known) mean λ.

To answer our question: *how many times is the primitive invoked in a given time interval*, we need to sample the (Poisson) probability distribution. That is, we need to find X, given the value of x, a random variable, and given the Poisson parameter λ.

The normal way of sampling a probability distribution is to compute the cumulative distribution function $F(x)$:

$$F(x) = P(X \leq x) = p = \int_{-\infty}^{b} f(x)dx \quad (4)$$

and then solve for x, given a (usually uniform) random variable p (generated typically using a random number generator). This can be awkward to do since there is not always an explicit closed form expression for $F(x)$.[7]

In the case of the Poisson distribution, we will adopt the approach given in [45] which computes the value of the random sample X iteratively.

We have already noted that, if events occur at random in time, then the number of events in a fixed interval of time has a Poisson distribution. We note now that for such random events, the time interval *between* events has an exponential distribution $f(x)$, where

$$f(x) = \lambda e^{-\lambda x}, \qquad \lambda > 0 \quad (5)$$

This distribution has a cumulative distribution $F(x)$, where

$$F(x) = p = 1 - e^{-\lambda x} \quad (6)$$

To find the inverse distribution, we solve for x:

$$x = \frac{-ln(1-p)}{\lambda} \quad (7)$$

Thus, given a uniform random variable p (*i.e.* a random number), we can compute the random sample from the exponential distribution straight-forwardly as $x = \frac{-ln(1-p)}{\lambda}$. Usually, we use the expression

$$x = \frac{-ln(p)}{\lambda} \quad (8)$$

since if $1 - p$ is a uniform random variable then so too is p.

The idea is to find how many exponential events (*i.e.* random samples of the exponential distribution) just fit into a unit time interval. That number will have a Poisson distribution: its value will be the required (Poisson) random sample.

In detail, let u_i be a uniform random variable and let v_i be an exponential random variable. We find v_i by sampling the exponential distribution. That is, v_i is given by:

$$v_i = \frac{-\ln(u_u)}{\lambda} \quad (9)$$

To find how many exponential events X fit in one time interval, we try to find the value of X (which has a Poisson

7. Strictly speaking, this approach applies only for continuous probability distributions.

distribution with parameter λ if we sample from an exponential distribution with parameter λ) such that

$$\sum_{i=0}^{X-1} v_i < 1 \leq \sum_{i=0}^{X} v_i, \qquad X = 0, 1, 2, \ldots \tag{10}$$

That is, the sum of the intervals between $X-1$ arrivals and X arrivals bounds one time interval. Thus, the Poisson random sample X we require is given by:

$$\sum_{i=1}^{X} \frac{-ln(u_i)}{\lambda} < 1 \leq \sum_{i=1}^{X+1} \frac{-ln(u_i)}{\lambda}, X = 0, 1, 2, \ldots \tag{11}$$

That is, X events just fit in one time interval. Multiplying across by $-1/\lambda$

$$\sum_{i=1}^{X} -ln(u_i) > -\lambda \geq \sum_{i=1}^{X+1} ln(u_i) \tag{12}$$

The sum of logs equals the log of products, hence

$$\ln \prod_{i=0}^{X-1} u_i > -\lambda \geq \ln \prod_{i=0}^{X} u_i \tag{13}$$

Exponentiating:

$$\prod_{i=0}^{X-1} u_i > e^{-\lambda} \geq \prod_{i=0}^{X} u_i \tag{14}$$

This expression essentially says that for two successive values of X (*e.g.* $X-1$ and X, $X = 0, 1, 2, \ldots$), $u_0 u_1 u_2 \ldots u_{X-1} > e^{-\lambda}$ and $u_0 u_1 u_2 \ldots u_X \leq e^{-\lambda}$. This gives us the basis for an iterative procedure to compute X by taking progressively more random numbers (u_i) and multiplying them together until we find a product that is less that $e^{-\lambda}$.

Specifically, we take a uniform random variable u_0 (using a random number generator). If $u_0 \leq e^{-\lambda}$, we stop and take $X = 0$. If $u_0 > e^{-\lambda}$, we generate another random number u_1, form the product $u_0 u_1$, and check again. If $u_0 u_1 \leq e^{-\lambda}$, take $X = 1$. We continue in this way until we find x such that $u_0 u_1 u_2 \ldots u_x$ is the first product $\leq e^{-\lambda}$, and we then take $X = x$. We state this in algorithmic form in Listing 2.

Listing 2. C function for sampling a Poisson distribution

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <math.h>

/* Generate a random sample from a Poisson distribution with a given mean, lambda
*/
/* Use the function rand() to generate a random number
*/

int samplePoisson(double lambda) {

   static bool first_call = true;
   int count;
   double product;
   double zero_probability;

   /* Seed the random-number generator with current time so */
   /* that the numbers will be different every time we run  */

   if (first_call) {
      srand( (unsigned)time( NULL ));
      first_call = false;
   }

   count = 0;
   product = (double) rand() / (double) RAND_MAX;

   zero_probability = exp(-lambda);

   while (product > zero_probability) {
      count++;
      product = product * ((double) rand() / (double) RAND_MAX);
   }
   return(count);
}
```

REFERENCES

[1] D. Brugali and E. Prassler, "Software engineering for robotics," *IEEE Robotics and Automation Magazine*, pp. 9, 15, March 2009. 1

[2] G. S. Broten, D. Makay, S. P. Monckton, and J. Collier, "The robotics experience — beyond components and middleware," *IEEE Robotics and Automation Magazine*, pp. 46–54, March 2009. 1

[3] D. Vanthienen, M. Klotzbücher, and H. Bruyninckx, "The 5C-based architectural Composition Pattern: lessons learned from re-developing the iTaSC framework for constraint-based robot programming," *Journal of Software Engineering for Robotics*, vol. 5, no. 1, pp. 17–35, 2014. 1, 4

[4] G. T. Heineman and W. T. Council, *Component-Based Software Egineering: Putting the pieces Together*. Reading, Massachusetts: Addison-Wesley, 2001. 1

[5] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. Reading, Massachusetts: Addison-Wesley, 2002. 1

[6] D. Brugali and P. Scandurra, "Component-Based Robotic Engineering (Part I)," *IEEE Robotics and Automation Magazine*, pp. 84–96, December 2009. 1, 3

[7] D. Brugali and A. Shakhimardanov, "Component-Based Robotic Engineering (Part II)," *IEEE Robotics and Automation Magazine*, pp. 100–112, March 2010. 1, 3

[8] "http://www.omg.org." 1, 4

[9] G. Metta, P. Fitzpatrick, and L. Natale, "Yarp: yet another robot platform," *International Journal on Advanced Robotics Systems*, vol. 3, no. 1, pp. 43–48, 2006. 1, 5

[10] P. Fitzpatrick, E. Ceseracciu, D. E. Domenichelli, A. Paikan, G. Metta, and L. Natale, "A middle way for robotics middleware," *Journal of Software Engineering for Robotics*, vol. 5, no. 2, pp. 42–49, 2014. 1

[11] "http://www.dream2020.eu." 2

[12] A. Tapus, A. Peca, A. Aly, C. Pop, L. Jisa, S. Pintea, and D. O. David, "Children with autism social engagement in interaction with nao, an imitative robot — a series of single case experiments," *Interaction Studies*, vol. 13, no. 3, pp. 315–347, 2012. 2

[13] B. Vanderborght, R. Simut, J. Saldien, C. Pop, A. S. Rusu, S. Pineta, and D. O. David, "Using the social robot probo as a social story telling agent for children with asd," *Interaction Studies*, vol. 13, no. 3, pp. 348–372, 2012. 2

[14] B. Scassellati, H. Admoni, and M. Mataric, "Robots for use in autism research," *Annual Review of Biomedical Engineering*, vol. 14, pp. 275–294, 2012. 2

[15] H. Kozima, C. Nakagawa, and Y. Yasuda, "Interactive robots for communication-care: A case-study in autism therapy," in *Proc. IEEE International Workshop on Robot and Human Interactive Communication (ROMAN)*, Nashville, TN, 2005. 2

[16] D. David, S. A. Matu, and O. A. David, "Robot-based psychotherapy: Concepts development, state of the art, and new directions," *International Journal of Cognitive Therapy*, vol. 7, no. 2, pp. 192–210, 2014. 2

[17] S. Thill, C. Pop, T. Belpaeme, T. Ziemke, and B. Vanderborght, "Robot-assisted therapy for autism spectrum disorders with (partially) autonomous control: Challenges and outlook," *Paladyn*, vol. 3, no. 4, pp. 209–217, 2012. 2

[18] M. Y. Jung, M. Balicki, A. Deguet, R. H. Taylor, and R. Kazanzides, "Lessons learned from development of component-based medical robot systems," *Journal of Software Engineering for Robotics*, vol. 5, no. 2, pp. 25–41, 2014. 3

[19] D. Alonso, C. Vincente-Chicote, F. Ortiz, J. Pastor, and B. Alvarez, "V^{3}CMM: a 3-view component meta-model for model-driven robotic software development," *Journal of Software Engineering for Robotics*, vol. 1, no. 1, pp. 3–17, 2010. 3

[20] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666—677, 1978. 3

[21] H. Bruyninckx, M. Klotzbücher, N. Hochgeschwender, G. Kraetzschmar, L. Gherardi, and D. Brugali, "The BRICS component model: A model-based development paradigm for complex robotics software systems," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, ser. SAC '13. New York, NY, USA: ACM, 2013, pp. 1758–1764. 1, 3, 4

[22] H. Bruyninckx, "Robotics software framework harmonization by means of component composability benchmarks," BRICS Deliverable D8.1, 2010, http://www.best-of-robotics.org. 3

[23] P. Soetens, H. Garcia, M. Klotzbücher, and H. Bruyninckx, "First established CAE tool integration," BRICS Deliverable D4.1, 2010, http://www.best-of-robotics.org. 4

[24] A. Shakhimardanov, J. Paulus, N. Hochgeschwender, M. Reckhaus, and G. K. Kraetzschmar, "Best practice assessment of software technologies for robotics," BRICS Deliverable D2.1, 2010, http://www.best-of-robotics.org. 4

[25] M. Radestock and S. Eisenbach, "Coordination in evolving systems," in *Trends in Distributed Systems, CORBA and Beyond*. Springer, 1996, pp. 162—176. 4

[26] C. Schlegel, A. Steck, and A. Lotz, "Model-driven software development in robotics: Communication patterns as key for a robotics component model," in *Introduction to Modern Robotics*. iConcept Press, 2011. 4, 5

[27] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng, "Ros: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009. 5

[28] J. Baillie, "Urbi: towards a universal robotic low-level programming language," in *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2005*, 2005, pp. 3219—3224. 5

[29] P. Soetens, "A software framework for real-time and distributed robot and machine control," Ph.D. dissertation, Department of Mechanical Engineering, Katholieke Universiteit Leuven, Belgium, 2006. 5

[30] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Oreback, *Software Engineering for Experimental Robotics*, ser. Springer Tracts in Advanced Robotics. Springer, 2007, ch. Orca: a component model and repository, pp. 231—251. 5

[31] A. Makarenko, A. Brooks, and T. Kaupp, "On the benefits of making robotic software frameworks thin," in *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2007*, 2007. 5

[32] E. I. Barakova, J. C. C. Gillesen, B. E. B. M. Huskens, and T. Lourens, "End-user programming architecture facilitates the uptake of robots in social therapies," *Robotics and Autonomous Systems*, vol. 61, pp. 704–713, 2013. 5

[33] D. Vernon, G. Sandini, and G. Metta, "The iCub cognitive architecture: Interactive development in a humanoid robot," in *Proceedings of IEEE International Conference on Development and Learning (ICDL)*, Imperial College, London, 2007. 6

[34] D. Vernon, C. von Hofsten, and L. Fadiga, *A Roadmap for Cognitive Development in Humanoid Robots*, ser. Cognitive Systems Monographs (COSMOS). Berlin: Springer, 2010, vol. 11. 6

[35] D. Vernon, *Artificial Cognitive Systems — A Primer*. Cambridge, MA: MIT Press, 2014. 6

[36] C. Schlegel, "Communication patterns as key towards component-based robotics," *International Journal of Advanced Robotics Systems*, vol. 3, no. 1, pp. 49–54, 2006. 6

[37] GNU coding standards. [Online]. Available: www.gnu.org/prep/standards/ 7

[38] Java code conventions. [Online]. Available: www.oracle.com/technetwork/java/codeconvtoc-136057.html 7

[39] C++ coding standards. [Online]. Available: http://www.possibility.com/Cpp/CppCodingStandard.html 7

[40] D. van Heesch. (2005) Doxygen user manual. [Online]. Available: www.doxygen.org 7

[41] D. Vernon, "Software engineering standards," DREAM Deliverable D3.2, 2014, http://www.dream2020.eu. 7.3

[42] ——, "Quality assurance procedures," DREAM Deliverable D3.3, 2014, http://www.dream2020.eu. 7.5

[43] M. Klotzbücher and H. Bruyninckx, "Coordinating robotics tasks and systems with rFSM statecharts," *Journal of Software Engineering for Robotics*, vol. 1, no. 3, pp. 28–56, 2012. 8.3

[44] A. Paikan, P. Fitzpatrick, G. Metta, and L. Natale, "Data flow port monitoring and arbitration," *Journal of Software Engineering for Robotics*, vol. 5, no. 1, pp. 80–88, 2014. 8.3

[45] D. Cooke, A. H. Craven, and G. M. Clarke, *Statistical Computing in Pascal*. London: Edward Arnold, 1984. 8.3, 9, 9

David Vernon is Professor of Informatics at the University of Skövde, Sweden. His interests are cognitive robotics and computer vision, focusing on cognitive architectures, cognitive development, and models of autonomy. He is a Senior Member of the IEEE, a Chartered Engineer of the Institution of Engineers of Ireland, and a past Fellow of Trinity College Dublin. He is the DREAM project's research director and main responsible for system integration. Previously, he was the technical coordinator of the EU-funded RobotCub project to develop the iCub humanoid robot. Prior to embarking on his university career, he was a software engineer with Westinghouse Electric.

Erik Billing is a a researcher in the Interaction Lab, School of Informatics, University of Skövde, Sweden. His research involves robot learning, planning of goal directed actions, modeling various aspects of human and animal cognition, and development of therapeutic tools for children with autism spectrum disorder. He is also the head of SweCog, the Swedish Cognitive Science Society.

Paul Hemeren is a Senior Lecturer in Cognitive Science and Head of Division in the School of Informatics, University of Skövde, Sweden. He received his Ph. D. from the University of Lund for a thesis on human action representation and the perception of biological motion. His main research interests are in human and robot action perception and understanding.

Serge Thill is an Associate Professor of Cognitive Science at the University of Skövde, Sweden and head of the Interaction Lab in the School of Informatics. He received his Ph. D. from the University of Leicester, UK. His main research interest is the modelling of the mechanisms underlying human embodied cognition and social interaction with applications to cognitive robotics.

Tom Ziemke is Professor of Cognitive Science at the University of Skövde, Sweden and a Visiting Professor in Cognitive Systems in the Dept. of Computer and Information Science (IDA) at Linköping University. He received his Ph. D. from the University of Sheffield, UK. His main research interest are embodied cognition and interaction, i.e. the role the body plays in cognitive and emotional processes, in social interactions, and people's interactions with different types of technology. He coordinates the DREAM project as well as a Swedish research initiative called AIR on action and intention recognition in human interaction with autonomous systems.