ENACT: An Efficient and Extensible Entity-Actor Framework for Modular Robotics Software Components

Werner, Tobias
tobias.werner@uni-bayreuth.de

Gradmann, Michael
michael.gradmann@uni-bayreuth.de

Orendt, Eric M.
eric.orendt@uni-bayreuth.de

Sand, Maximilian
maximilian.sand@uni-bayreuth.de

Spangenberg, Michael
michael.spangenberg@uni-bayreuth.de

Henrich, Dominik
dominik.henrich@uni-bayreuth.de

Chair for Robotics and Embedded Systems, Fax (+49) 0921/55-7682
Universität Bayreuth, D-95440 Bayreuth, Germany

Abstract

Recent advances towards generic robot programs require efficient sharing of copious information among many functional modules. Current solutions favor extensibility over efficiency and thus limit attainable functionality. In contrast, we contribute a software framework that efficiently implements the extensible entity-aspect-actor paradigm. Our framework remains safe for multi-threading, scales to distributed systems, handles inconsistent data and supports history logging. Evaluation through scenarios in industrial and service robotics attests claimed benefits. We conclude that our framework is fit for use in robotics, with significance also for computer games, simulations, and VR applications.

1 Introduction

Over the past decades, general purpose robot manipulators have become a vital part of industrial automation due to their precision and strength. Recent research aims at extending these benefits past traditional work cells and past pre-programmed trajectories. Apart from enhancing industrial capabilities, this opens up robot manipulators to attractive use cases in the service sector, in small and medium-sized enterprises, and in home use.

To support extended use cases, robots must perceive and understand their surroundings, and they must respond in a reasonable and timely manner. In terms of software, an extensible set of complex algorithms must efficiently interact to evaluate copious amounts of incoming data.

Interaction complexity between algorithms necessitates sophisticated data management. Current research suggests data storage and exchange through knowledge databases, system architectures or world models, each with individual benefits and shortcomings. For example, knowledge databases hold extensible relational data, but do not efficiently scale to subsymbolic content.

In contrast, we contribute a framework for data management over modular robot software components that unites benefits of existing approaches while avoiding common pitfalls. Our entity-actor framework **ENACT** efficiently implements the extensible entity-actor-paradigm [4] to achieve weak coupling and strong cohesion over software components. We process subsymbolic and symbolic information alike, and we offer multiple backing locations for opaque data distribution or history logging. Finally, the slim programming interface of ENACT is inherently thread-safe and multilock-capable, yet without the overhead of popular copy-on-write mechanisms. See Figure 1 for select data available through our framework.

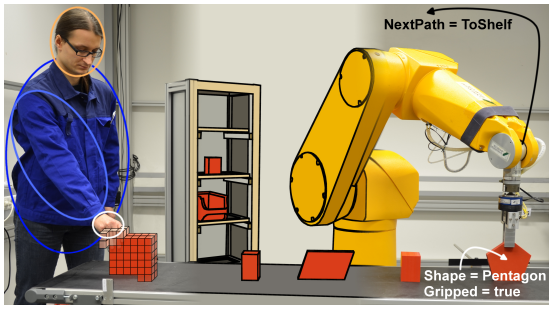

Figure 1: Our framework efficiently stores and connects symbolic and subsymbolic data for human-robot cooperation in an industrial use case. Left to right: Limb approximations, voxel reconstruction, CAD models, high-level properties.

This paper is structured as follows: Section 2 investigates related knowledge databases, system architectures and world models. For each candidate, we report respective advantages and shortcomings. In Section 3, we introduce the ENACT software framework as an improvement over identified shortcomings. We formally specify framework components, we examine algorithm interaction, and we elaborate vital low-level features. Throughout Section 4, we evaluate benefits of our framework with two use cases from service and industrial robotics. Section 5 concludes our contribution with a critical discussion of ENACT and an outlook on future work.

2 Related Work

Related research separates into three major categories: knowledge databases, robot system architectures, and world models.

Knowledge databases (e.g. [5], [11], [13]) are foremost concerned with structuring and interpreting interwoven

information. Data choice in general is not restricted, although typical implementations must work with symbolic information at coarse abstraction levels to derive meaningful statements. Usually, knowledge databases do not optimize for efficiency or latency, and thus are not suited for applications that require high responsiveness.
RoboBrain [11] is a typical example for a knowledge database. Notably, RoboBrain manages knowledge in a content-rich graph, where nodes store ground-truth facts and where annotated edges carry relations between these facts. A custom language for template matching supports generic, albeit expensive, database requests.
Robot system architectures (e.g. [1], [3], [8]) suggest a structuring for software components in robotics systems. This structuring groups software components into high-level software layers and establishes communication channels between these layers. Focus points of individual architectures include distributed processing, system reliabilty, or process scheduling, in general with a bias for high-level design over low-level performance.
For example, the renowned ROS (Robot Operating System) framework [8] intuitively connects distributed ROS nodes. Yet, the specific design of ROS with its message passing and publisher-subscriber paradigms becomes a bottleneck when sharing complex remote world states.
World models (e.g. [2], [9], [15]) strive for a balance between knowledge databases and system architectures by considering efficient storage and transfer of extensible low-level data. For instance, the scene graph of [2] only stores physical relations between individual geometric object representations and thus avoids the overhead of generic communication channels or graph structures.
Note that the term "world model" is not clearly defined: Common world models in robotics hold geometric data only (e.g. triangles, voxels), while typical world models in interactive simulations and computer games also include low-level relational knowledge. A standard world model for robotics applications has yet to emerge.

3 The ENACT framework

This section introduces components within the ENACT framework formally and describes their interactions.
We proceed in a bottom-up manner: First, we specify static components of the framework. We then continue by describing dynamic interactions between components. Finally, we derive relevant implementation features from components and their interactions.

3.1 Static Components

Entities and their aspects form the main components of the ENACT framework:
Entities represent objects of interest to an individual application, and usually map to objects in the physical world. The choice of entities depends on the application scenario. For example, a work piece might be a single entity in a workspace surveillance setting, while the same work piece might decompose into multiple entities for an assembly task. In the following, we denote the set of all $|E|$ world entities e_i for a specific application as

$$E = \{e_1,\ e_2,\ e_3,\ ...,\ e_{|E|}\}.$$

Aspects demarcate global property classes of all entities. As with entities, the application dictates the selection of aspects. Aspects range from shape representations (point clouds, triangles) over physical parameters (mass, color) to derived knowledge-like attributes (placement relations, object affordance). We refer to the set of all $|A|$ aspects a_j for a specific application as

$$A = \{a_1,\ a_2,\ a_3,\ ...,\ a_{|A|}\}.$$

Entities and aspects are static and cannot change at runtime. This constraint helps with thread-safety in the later implementation. In this context, note that neither entities nor aspects directly store any data payload.

3.2 Dynamic Components

A separate mechanism implements the actor paradigm to update data associated with entities and aspects through one or more explicit world contexts:
World contexts store a single datum for each aspect of each entity. For instance, a world context might hold the datum "5 kg" for the mass aspect of a work piece entity. Formally, each aspect a_j requires data from a set D_j, and each entity e_i has a value $d_{i,j} \in D_j$ for each aspect a_j. Therefore, a world context W is a map

$$W :\ E \rightarrow D_1 \times D_2 \times D_3 \times ... \times D_{|A|},$$
$$W(e_i) = (d_{i,1},\ d_{i,2},\ d_{i,3},\ ...,\ d_{i,|A|}).$$

Applications can select custom data sets for their aspects. Thereby, our framework becomes intuitive to extend.
Actors form the only interactive components in ENACT and update world contexts over time. For example, world context data may change due to sensor readings or user interaction. Actors accordingly range from data sources (sensors, cameras) over algorithms (path planning, object recognition) to data sinks (robot manipulators).
In formal terms, each actor $\mathfrak{a}_k$ from a set of $|\mathfrak{A}|$ actors

$$\mathfrak{A} = \{\mathfrak{a}_1,\ \mathfrak{a}_2,\ \mathfrak{a}_3,\ ...,\ \mathfrak{a}_{|\mathfrak{A}|}\}$$

works on a set of entities $E_{\mathfrak{a}_k} \subset E$ and their aspects through some update logic $U_{\mathfrak{a}_k}$.
Actors execute over intervals of time $\mathbb{T}$, ideally in parallel to increase efficiency. With locking and waiting, there always is at least a single conflict-free sequential execution path: For any time $t \in \mathbb{T}$, the set of active actors $\{\mathfrak{a}_{t,1}, \mathfrak{a}_{t,2}, \mathfrak{a}_{t,3}, ...\}$ thus satisfies

$$E_{\mathfrak{a}_{t,x}} \cap E_{\mathfrak{a}_{t,y}} = \emptyset \qquad \text{for } x \neq y.$$

The world context W_{t_e} at time $t_e \in \mathbb{T}$ then derives from its directly preceding context W_{t_s} at time $t_s \in \mathbb{T}$ as:

$$W_{t_e}(e_i) = \begin{cases} U_{\mathfrak{a}_{t_e,c}}(W_{t_s}, e_i) & \text{if } \exists \mathfrak{a}_{t_e,c} \text{ with } e_i \in E_{\mathfrak{a}_{t_e,c}}, \\ W_{t_s}(e_i) & \text{otherwise,} \end{cases}$$

with some default initial context for $t_s = t_0$.

Finally, consider the deliberate assignment of separate variables for consecutive world contexts: While we reuse unchanged world data to avoid expensive copies, we can still maintain multiple separate world contexts. In turn, this allows us to to model inconsistent world views, distributed data storage or data history. Figure 2 illustrates a configuration with two world contexts.

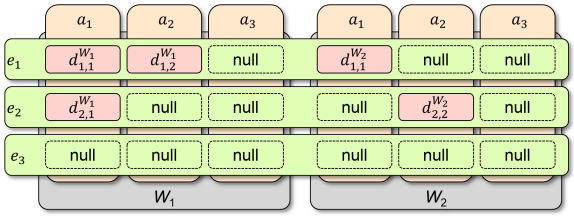

Figure 2: World contexts W_1, W_2 hold data for entities e_1 to e_3 and aspects a_1 to a_3. Aspect data sets D_j contain a **null** marker to indicate unknown information: W_2 does not know a value for aspect a_2 on entity e_1, and neither W_1 nor W_2 stores data for a_3 or e_3. Finally, aspect a_1 on entity e_1 has potentially inconsistent data $d_{1,1}^{W_1}$ and $d_{1,1}^{W_2}$.

3.3 Key Implementation Features

In contrast to popular high-level framework designs, we built the ENACT framework around low-level efficiency. The following paragraphs highlight notable features in our low-level C++11 implementation and describe their ties to the above formal specification.

In a typical setup, a client application initializes through instantiating one or more world contexts. Thereafter, the client application creates multiple actors, one for each dynamic component required by the application scenario. The programming interface of the ENACT framework already implements fundamental classes for world contexts and basic actors. In particular, most actors use an internal thread to update a construction-bound world context.

Actors identify unique entities and unique aspects by allocating generic ID objects on the program heap, one ID per $e_i \in E$ or $a_j \in A$. Unique address allocation through the C++ runtime guarantees unique IDs without the management overhead and implicit risk of popular integer-based ID counters. Shared pointers from the C++11 standard library realize lifetime management on ID objects. This enables us to automatically and efficiently track ID lifetime over thread boundaries.

After creating entity and aspect IDs, actors can register these IDs on world contexts to apply initial context data. We use unique pointers for transfer of initial data, as ownership transfer semantics prohibit unlocked data access after initial registration. For example, an actor could register a work piece entity with a mesh aspect as follows:

```
shared_ptr<entity_id> e_wpiece(new entity_id());
shared_ptr<aspect_id> a_mesh(new aspect_id());
unique_ptr<mesh> m(load_file("phone.ply"));
world_context.init(e_wpiece, a_mesh, move(m));

// m is NULL now.
// Future access only possible through lock.
```

To guarantee robust threaded access, our IDs do not carry individual data. Instead, we enforce an intuitive synchronization mechanism based on the C++ RAII (resource acquisition is initialization) pattern. Actors must explicitly retrieve individual data from a world context by creating lock objects on the program stack. Locks therefore automatically disengage on method exit, even in case of any exception. Each thread is allowed but a single lock at a time, otherwise the lock constructor throws an exception. Since each lock performs an atomic operation even for multiple IDs, this setup is implicitly free of race conditions or deadlocks. Finally, world contexts globally track locked data. This enables us to realize efficient reader-writer locks for improved parallelism. With the ENACT templates for reader-writer locks and data access, an actor might update the object count on a workbench as such:

```
list<lock_request> lock_requests
  ({ lock_request(e_wpiece, a_mesh, readonly),
     lock_request(e_wbench, a_ocount, write) });

lock l(world_context, lock_requests);
const_access<mesh> m(l[0]);
access<int> ocount(l[1]);
if (m->bbox.minz < 0) { (*ocount) += 1; }
```

World contexts in our initial implementation only carry local data. We can therefore provide efficient and copy-less access to large-scale data through references in the lock objects. However, our framework intuitively scales to distributed world contexts, where the locking mechanism can opaquely fetch data from a remote source.

While we use C++ as our language of choice, implementation details map naturally to other programming languages. For instance, our framework ports to Java with just minor adaptions: While Java garbage collection allows us to drop shared pointers in favor of direct ID references, we must replace C++ RAII paradigms with Java's less sophisticated try-catch-finally mechanisms.

4 Evaluation

We evaluate the ENACT framework in two application scenarios from the field of industrial and service robotics: One application demonstrates the low-latency sharing of copious world context data over multiple threads, while the other application emphasizes the intuitive integration of formerly separate software components.

4.1 Low-Latency Application

Our first example application addresses the domain of industrial human-robot cooperation. In this application, we monitor a shared human-robot workspace through a multi-camera network. As shown in Figure 3, we derive visual hulls of workspace obstacles from segmented camera images. In the end, we intend to use these visual hulls to enable collision avoidance and distance-based speed control for a general-purpose robot manipulator. See [14] for further details on this setup.

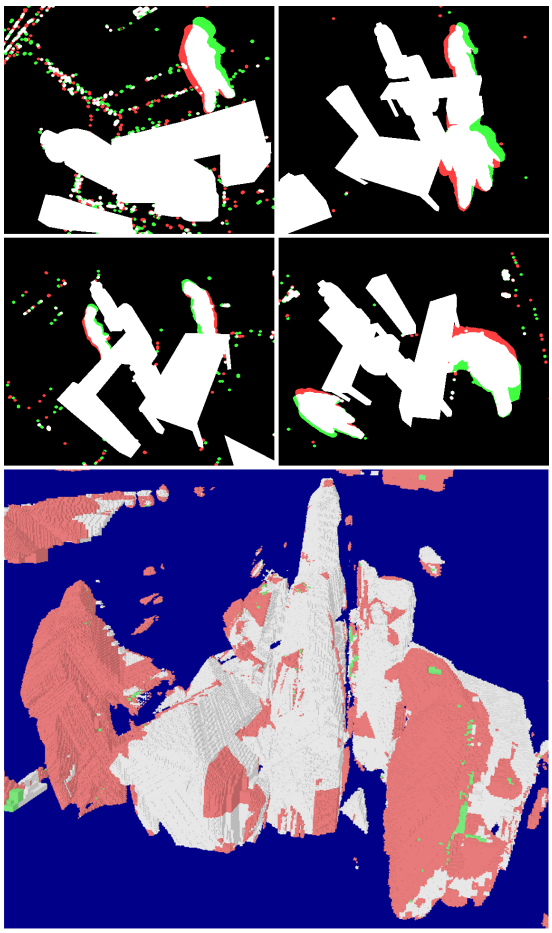

Figure 3: Top: Images from the multi-camera system after foreground-background segmentation. Bottom: A visual hull reconstruction before knowledge-based refinement. Only colored pixels and voxels have changed over the preceding frame.

In terms of ENACT, the multi-camera system translates to a series of entity-actor pairs. For each camera, a separate **camera actor** fetches incoming images within an internal thread. All camera actors subsequently store their images to a global world context under a camera-specific entity ID and with an aspect ID that demarcates camera images. Finally, **background subtraction actors** convert stored camera images to segmented silhouettes.

Silhouettes form the input to the **reconstruction actor**. This actor builds a visual hull over all silhouettes, rejects false-positive obstacle detections with knowledge-based refinement, and attaches remaining obstacle geometry to a workspace entity. In turn, the workspace entity passes through a real-time **path planning actor**. In the end, path planning attaches speed and path aspects to a robot entity for highly responsive execution by the **robot actor**.

Actors for geometric shape reconstruction and human tracking intuitively integrate into the above basic setup: A **shape reconstruction actor** (e.g. [10]) can provide boundary representations of static workspace obstacles through an offline step, while a **human tracking actor** (e.g. [6]) enables recognizing and predicting humans within the workspace. World context data contributed by either actor is readily available to path planning.

Figure 4 gives an overview over all entities, aspects, and actors involved in the first example application.

The challenge of the first example application lies in low-latency demands of collision-free path planning. For our evaluation, we run the application on a midrange 2015 eight-core i7 system, using eight Logitech C930e USB cameras on VGA resolution, with a target 10 Hz update rate for path planning and speed control. The ENACT framework enables us to efficiently load all cores without error-prone individual locking strategies. At the same time, the specific impact of ENACT on overall system load remains well below measuring tolerances.

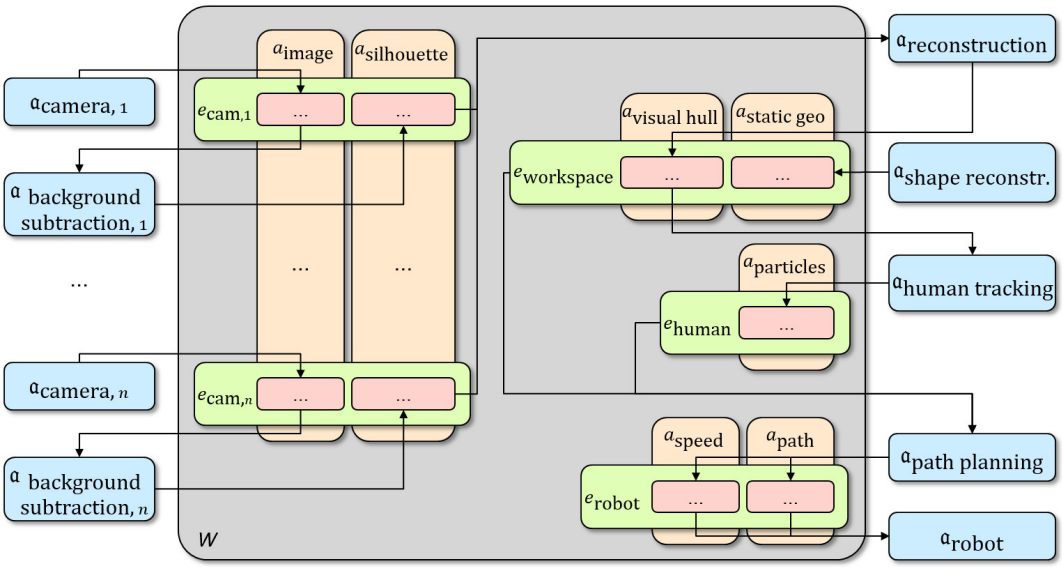

Figure 4: Overview over all entities (green), aspects (orange), and actors (blue) involved in the low-latency example application. All relevant data is located in a single world context (gray).

4.2 Integrative Application

For our second example application, we examine a robot-automated palletizing task. In this task, a conveyor belt delivers various food packages to a KUKA Lightweight Robot (LBR) for palletizing on a metal tray. The robot perceives its surroundings through an overhead depth sensor and employs a SCHUNK PG 70 servo-electric two-finger-parallel gripper to grasp incoming packages. Actual palletizing then requires two separate steps: First, the robot places the food package onto the tray. Second, the robot pushes the package until it comes into contact with already palletized packages. In rare cases, grasping or placement is inaccurate, and deformed food packages end up on the metal tray. We must detect such errors to enable appropriate error handling. Figure 6 illustrates the overall task setup and a potential error condition.

We realize error handling through a two-step process: The first step generates offline ground-truth geometry for each type of food package, while the second step uses the ground-truth geometry for online screening of incoming food packages.

To generate ground-truth data in the first step, we employ another KUKA LBR. This robot applies an eye-in-hand depth sensor to acquire images of some food package over varying viewpoints. From these images, we derive a geometric reconstruction of the package. See Figure 5 for example ground-truth surface models.

Opposed to human-robot collaboration from the previous example, palletizing has less rigid timing requirements. As challenge for the ENACT framework, the palletizing example instead involves a variety of collaborating software components, including perception, reconstruction, sensor-based robot control, and error monitoring. In particular, we already had developed most components in isolation for other applications. We consequentially had to integrate the components into ENACT to accomplish the palletizing task. The extensible, modular architecture of ENACT enables this integration with just minor effort. In the following, we detail the interaction of resulting actors, entities, aspects and world contexts.

Figure 5: Boundary representation models of two example objects as reconstructed in the offline step of error handling.

The **perception actor** controls the vision system. In the offline step, perception initializes entities $e_{view,i}$ for each viewpoint i while the eye-in-hand robot orbits a food package. Perception also registers two aspects for each created entity. One aspect holds a point cloud as retrieved from the depth sensor, while the other aspect stores the acquisition pose. Once perception has finished an entity $e_{view,i}$, the **reconstruction actor** creates a boundary representation model from the point cloud and attaches the model to another aspect. A **fusion actor** finally combines all individual viewpoint models $e_{view,i}$ to a complete surface model of the food package and stores the result in e_{object}. Details on reconstruction and fusion of boundary representations can be found in [10]. Note that all of the operations above use a world context W_{training} to compile ground-truth data for the following online step.

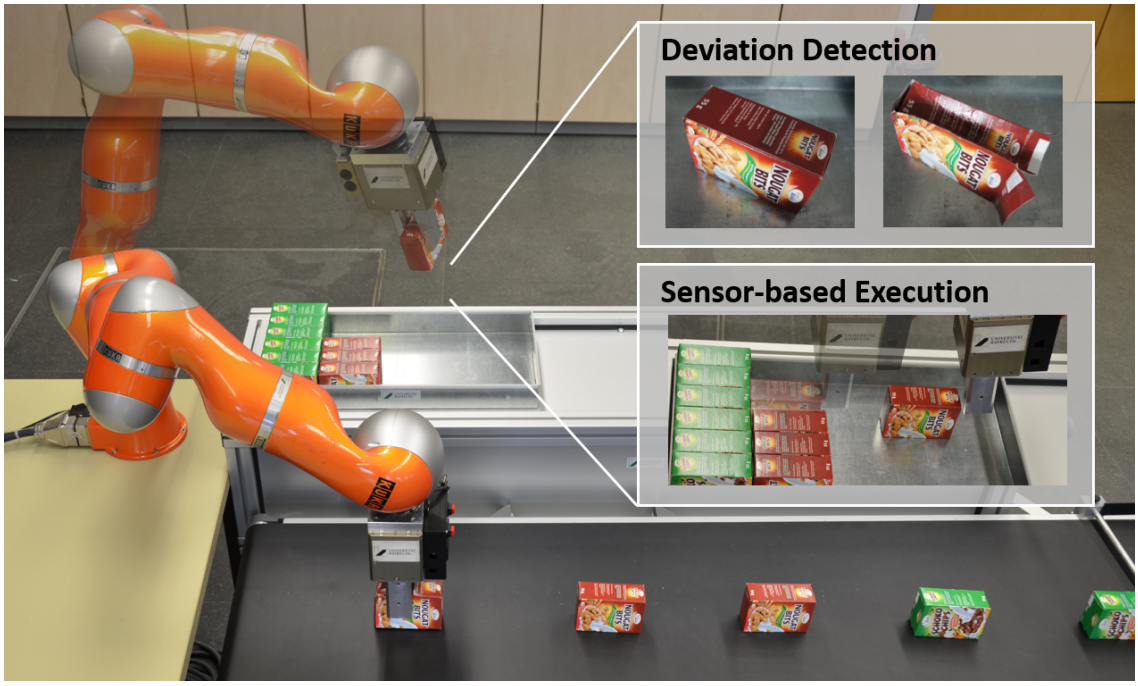

Figure 6: Components of the integrative application: A conveyor belt, incoming food packages, a robot with gripper, and a metal tray for palletizing the packages. The figure does not show the overhead depth sensor.

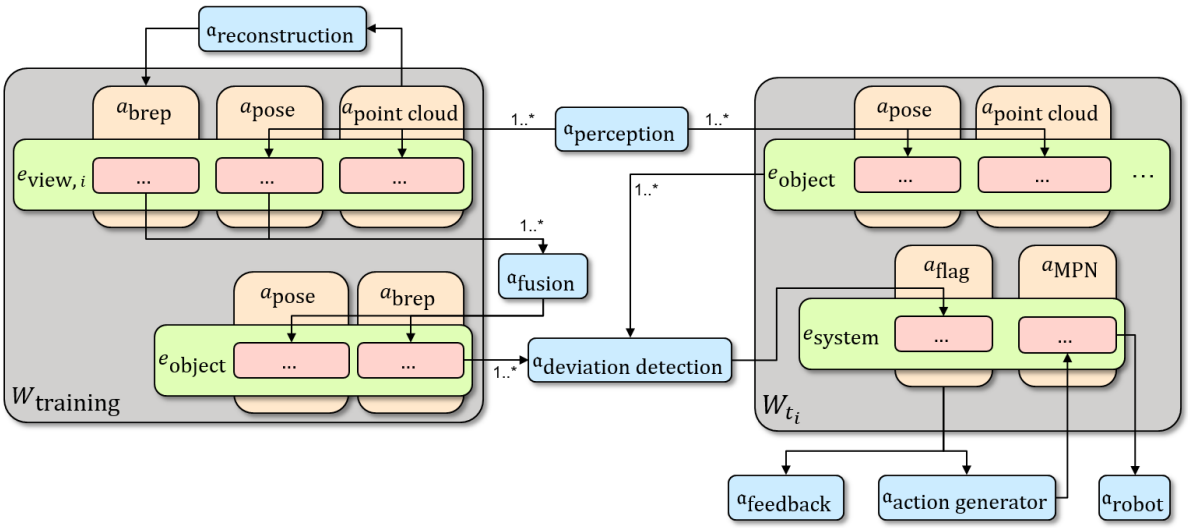

Figure 7: Overview over entities (green), aspects (orange) and actors (blue) of the integrative application. Our solution uses two separate world contexts (gray): W_{training} stores ground-truth geometric data as generated in an offline step, while W_{t_i} models online execution and updates repeatedly.

Interactions in the online step also rely on the **perception actor** to handle the vision system. Namely, perception periodically updates all visual aspects (e.g. poses) for any entities e_{object}. Those entities represent real world objects in the robot workspace. When no existing entity matches incoming visual aspects, perception creates a new e_{object} instead. Resulting entities subsequently act as input both to object palletizing and to error monitoring.

For the task of object palletizing, the **action generator actor** parametrizes grounded action skeletons in the form of manipulation primitive nets (MPNs). The **action actor** then attaches the resulting MPN as an aspect to e_{system}. Finally, the **robot actor** executes the MPN in order to palletize the incoming object. Consider [12] for further information on grounding sensor-based actions.

Within error monitoring, the **deviation detection actor** compares entity instances e_{object} from W_{t_i} and W_{training}. First, the actor locates matching entities in both world contexts. Second, the actor compares visual aspects of matching entities for potential differences. Depending on specific differences between entities from W_{t_i} and W_{training}, the actor finally raises a flag aspect in e_{system} to signal an unexpected entity state. The **feedback actor** reacts to the flag in e_{system} and notifies the system user about the detected deviation. For further details on entity based deviation detection and classification, see [7].

See Figure 7 for an overview over all final entities and actors in this integrative application.

5 Conclusion

In the preceding, we have formally specified **ENACT**, our entity-actor framework for robotics systems. Unlike other approaches, our contribution supports both efficient and extensible interaction between software components through a thin layer for data exchange. We confirmed this claim in two example applications: One application used ENACT for thread-safe, low-latency communication over multiple threaded actors, while the other application intuitively integrated multiple existing components to solve an intricate automation task.

Apart from realizing distributed world contexts, future work consists of integrating additional components and applications into the ENACT framework. Finally, export of ENACT as a single ROS node remains an option for higher-level software integration.

References

[1] James S. Albus and Anthony J. Barbera: *RCS: A Cognitive Architecture for Intelligent Multi-Agent Systems*, Intelligent Autonomous Vehicles Conference (IAV), 2013.

[2] Blumenthal et al.: *A Scene Graph Based Shared 3D World Model for Robotic Applications*, International Conference on Robotics and Automation (ICRA), 2013.

[3] Firby et al.: *An Architecture for Vision and Action*, International Joint Conference on Artificial Intelligence (IJCAI), 1995.

[4] M. E. Latoschik and H. Tramberend: *A scala-based actor-entity architecture for intelligent interactive simulations*, Software Engineering and Architectures for Realtime Interactive Systems (SEARIS), 2012.

[5] Lim et al.: *Ontology-Based Unified Robot Knowledge for Service Robots in Indoor Environments*, IEEE Transactions on Systems, Man, and Cybernetics, 2011.

[6] Antje Ober-Gecks, Maria Hänel, Dominik Henrich, and Tobias Werner: *Fast multi-camera reconstruc-*

tion and surveillance with human tracking and optimized camera configurations, International Symposium on Robotics (ISR) / German Conference on Robotics (ROBOTIK), 2014.

[7] Eric M. Orendt and Dominik Henrich: *Design of Robust Robot Programs: Deviation Detection and Classification using Entity-based Resources*, IEEE International Conference on Robotics and Biomimetics (ROBIO), pp. 1704-1710, 2015.

[8] Quigley et al.: *ROS: an open-source Robot Operating System*, International Conference on Robotics and Automation (ICRA), 2009.

[9] Maayan Roth, Douglas Vail, and Manuela Veloso *A real-time world model for multi-robot teams with high-latency communication*, IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2003.

[10] Maximilian Sand and Dominik Henrich: *Incremental Reconstruction of Planar B-Rep Models from Multiple Point Clouds*, The Visual Computer, 2016 (to appear).

[11] Saxena et al.: *RoboBrain: Large-Scale Knowledge Engine for Robots*, arXiv:1412.0691, 2014.

[12] Michael Spangenberg and Dominik Henrich: *Grounding of actions based on verbalized physical effects and manipulation primitives*, IEEE International Conference on Intelligent Robots and Systems (IROS), pp. 844-851, 2015.

[13] Waibel et al.: *RoboEarth: A World Wide Web for Robots*, IEEE Robotics and Automation Magazine, pp. 69-82, 2011.

[14] Tobias Werner and Dominik Henrich: *Efficient and Precise Multi-Camera Reconstruction*, ACM/IEEE International Conference on Distributed Smart Cameras (ICDSC), 2014.

[15] Wurm et al.: *OctoMap: A Probabilistic, Flexible, and Compact 3D Map Representation for Robotic Systems*, International Conference on Robotics and Automation (ICRA), workshop on best practice in 3D perception and modeling for mobile manipulation. 2010