

Using the DiaSpec design language and compiler to develop robotics systems

Damien Cassou

Software Architecture Group, HPI
University of Potsdam, Germany

damien.cassou@hpi.uni-potsdam.de

Serge Stinckwich

UMI 209 UMMISCO
IRD/IFI/Vietnam National University

serge.stinckwich@ird.fr

Pierrick Koch

UMR 6072 GREYC
Université de Caen-Basse
Normandie/CNRS/ENSI-
CAEN

pierrick.koch@unicaen.fr

Abstract—A Sense/Compute/Control (SCC) application is one that interacts with the physical environment. Such applications are pervasive in domains such as building automation, assisted living, and autonomic computing. Developing an SCC application is complex because: (1) the implementation must address both the interaction with the environment and the application logic; (2) any evolution in the environment must be reflected in the implementation of the application; (3) correctness is essential, as effects on the physical environment can have irreversible consequences.

The SCC architectural pattern and the DiaSpec domain-specific design language propose a framework to guide the design of such applications. From a design description in DiaSpec, the DiaSpec compiler is capable of generating a programming framework that guides the developer in implementing the design and that provides runtime support. In this paper, we report on an experiment using DiaSpec (both the design language and compiler) to develop a standard robotics application. We discuss the benefits and problems of using DiaSpec in a robotics setting and present some changes that would make DiaSpec a better framework in this setting.

I. INTRODUCTION

A Sense/Compute/Control (SCC) application is one that interacts with the environment [17]. The SCC architectural pattern guides the description of SCC applications and involves four kinds of components, organized into layers [6], [10]: (1) *sensors* at the bottom, which obtain information about the environment; (2) then *context operators*, which process this information; (3) then *control operators*, which use this refined information to control (4) *actuators* at the top, which finally impact the environment. A robotics application is a kind of SCC application where the environment is composed of a robot (sensors/actuators/body, control architecture, etc) and the robot's neighborhood (the walls, ground, people, etc) [15]. As noticed by Taylor *et al.* [17], the Sense/Plan/Act architecture [15], widely used in robotics, closely resembles the SCC architectural pattern.

DiaSpec is a domain-specific design language dedicated to describing SCC applications [6], [7]. From such a design description, the DiaSpec compiler produces a dedicated Java programming framework that is both *prescriptive* and *restrictive*: it is prescriptive in the sense that it guides the developer, and it is restrictive in the sense that it limits the developer to what the design description allows. By separating application logic (implemented by the developers) and

runtime support (generated in the programming framework), DiaSpec facilitates the design, implementation and evolution of SCC applications.

Contributions

Our contributions are as follows:

- A *report* on an experiment of designing and implementing a standard robotics application in the SCC architectural pattern with the DiaSpec domain-specific design language and framework (Sections II and III). This report includes detailed instructions and guidelines to allow further experiments.
- A *discussion* of the benefits and problems of using DiaSpec in a robotics setting (Section IV). This discussion includes a list of changes to DiaSpec that would make it a better framework for developing new robotics applications.

We finally highlight some related works and conclude in sections V and VI.

II. DESIGNING A ROBOTICS APPLICATION

In this section we first explain how to decompose a robotics application in DiaSpec component types. Then we present a case study that we use as an example of how to describe a robotics application with DiaSpec.

In the rest of this paper we use ROS¹ as the underlying middleware for our case study. We believe it is a good choice as ROS is becoming a standard within the robotics community. It is important to note however that our approach and DiaSpec are independent of any middleware.

A. Decomposing

Designing an application with DiaSpec requires a decomposition in layers. Each layer corresponds to a separate type of component:

- A *sensor* sends information sensed from the environment to the context operator layer through data *sources*. A sensor can both push data to context operators and respond to context operator requests. We use the term “sensor” both for entities that actively retrieve information from the environment, such as system probes, and

¹<http://www.ros.org/wiki/>

entities that store information previously collected from the environment, such as structured information coming from the middleware.

- A *context operator* refines (aggregates and interprets) the information given by the sensors. Context operators can push data to other context operators and to control operators. Context operators can also respond to requests from parent context operators.
- A *control operator* transforms the information given by the context operators into orders for the actuators.
- An *actuator* triggers actions on the environment.

The following details the steps to follow to decompose a robotics application into these component types.

Reusing existing components: In the presence of a previous application developed with DiaSpec, it is possible and advisable to reuse as much components as possible. Depending on the amount of reused components this can have a huge impact on the application of the other steps.

Listing capabilities: Each robot comes with its own set of capabilities (e.g., sensing motion and projecting light). A developer should map these capabilities to sensor sources and actuator actions. A developer should then group related sources and actions inside *entity classes* (e.g., a camera providing a picture source and zooming action). Beside sources and actions, an entity class may also have *attributes* to characterize its instances (e.g., resolution, accuracy and status). In the presence of a high-level middleware (such as ROS), it can be useful to also map capabilities of the middleware into sources and actions (e.g., a mapping or a localization capability).

Identifying main context operators: The next step of the decomposition in components is the identification of the main high-level pieces of information required by the application. A developer should map these pieces of information to context operators and use them as input to control operators.

Decomposing into lower-level pieces: Then, a developer must identify lower-level context operators that act as input sources for the higher-level ones. This decomposition is typically done in several steps, each step slightly lowering the level of previously identified context operators. This decomposition ends when each identified context operator can directly take its input from a set of sensor sources.

Identifying control operators: From the high-level context operators a developer has to derive a set of control operators that will send orders to actuators. Because a developer can not reuse the code of a control operator in another part of the application, it is important that this code is as simple as possible. If there is opportunity for reuse, the code should be moved to a new context operator.

Identifying data types: While proceeding with the above steps it is also necessary to define data types. A developer then use these types to describe entity sources, context operators, and parameters of actuator actions. A data type is either primitive (e.g., integer, boolean and float), an enumeration of names (e.g., a luminosity can either be low, normal or high), or a structure (e.g., a coordinate with x and y fields). An important question arises in the presence

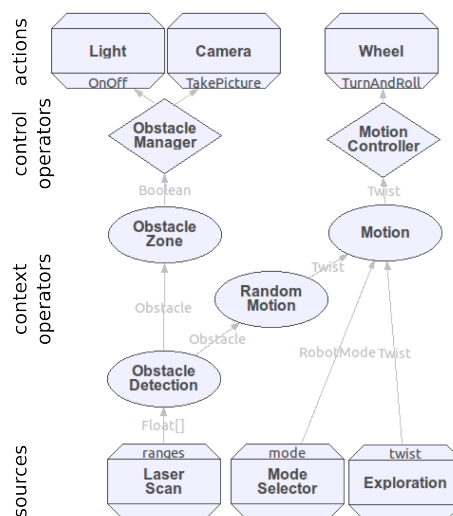


Fig. 1. The case study decomposed into the different type of components of DiaSpec

of a high-level middleware (such as ROS): should the types of the application be the types provided by the middleware or should the application define new types. The former solution is easier to use whereas the latter provides more decoupling. A general principle is to provide new types when their transcription in DiaSpec is straightforward (e.g., a coordinate) and to reuse the middleware types otherwise (e.g., ROS defines a “twist” data type that is complex enough to not be reimplemented).

B. Case Study

As a running example, we present an application that is typical of the robotics domain. In this application, a robot evolves in an unknown environment and has two modes: *random* and *exploration*. In the random mode, the robot goes straight and when an obstacle is close enough turns before going straight again. In the exploration mode, the robot goes to unvisited locations with the goal to visit as much as possible from the neighborhood. The current mode can be changed at anytime by an operator through a graphical interface. In both modes, the robot turns on an embedded projector and takes pictures when it is in a zone with obstacles.

Let us now discuss the above steps in the context of this case study (Figure 1 represents the result).

Reusing existing components: We assume no previous DiaSpec application and thus no DiaSpec component to reuse.

Listing capabilities: The Bosch robotics research group develops an *exploration* capability² based on a well-known frontier-based exploration algorithm [18]. In this algorithm the exploration is composed of two steps: a motion toward a location and a new observation of the environment at this location. The location is chosen among a set of candidate locations on the frontier between explored and unexplored

²<http://www.ros.org/wiki/explore>

space. This capability is exactly what we need for the exploration mode of the robot. Our robot comes with a range-finder type laser scanner, a light projector, a camera, and a set of wheels. From all these capabilities, we identify:

- a LaserScan entity class with a ranges source providing laser ranges from the sensor;
- A ModeSelector entity that provides a graphical interface for the operator to choose the current mode of the robot;
- an Exploration entity that provides a source of twists for the robot;
- a Light and Camera entities that respectively enlighten the neighborhood and take pictures on request;
- a Wheel entity that can turn or roll on request;

Identifying main context operators: The most important activity of our robot is to move. Therefore we introduce a Motion context operator that produces a twist, representing the motion of the robot. Because our robot takes pictures and turns on its projector when it is in a zone with obstacles, we introduce an ObstacleZone context operator that indicates whether or not some obstacles are in the neighborhood.

Decomposing into lower-level pieces: The Motion context operator produces a twist based on which mode is selected and on the twist values coming from both modes. The selected mode is directly provided by the ModeSelector entity. We introduce a RandomMotion context operator that produces twists for the random mode. The twist for the exploration mode is directly provided by the Exploration entity. Both the ObstacleZone and RandomMotion context operators need the information about nearby obstacles. We thus introduce the ObstacleDetection context operator to indicate the proximity of an obstacle.

Identifying control operators: The MotionController control operator takes information from the Motion context operator and transmits this information to the Wheel entity. The ObstacleManager control operator takes information from the ObstacleZone context operator and triggers the light and takes a picture with the camera.

Identifying data types: We have already seen that our application uses the notion of twist to indicate motion. A developer can define a twist as a pair of vectors which represent the linear and angular velocity. The robot current mode is represented as an enumeration of the RANDOM and EXPLORATION names. The ObstacleDetection context operator provides an Obstacle data type containing both a boolean to indicate if an obstacle is in front of the robot and a set of float numbers (the ranges) as provided by the laser scan giving details about the neighborhood.

C. Describing with DiaSpec

Once a developer decomposed the application using the different component types, the transcription to the DiaSpec design language is straightforward. Listing 1 gives an extract of the case study transcription.

In this listing, the entity, context, and controller keywords are respectively used to introduce

```

1 import Twist as org.ros.message.geometry_msgs.Twist;
2 structure Obstacle { isDetected as Boolean;
3   ranges as Float[]; }
4 enumeration RobotMode { RANDOM, EXPLORATION }
5 action OnOff { on(); off(); }
6 entity Light { action OnOff; }
7 entity LaserScan {
8   source ranges as Float[];
9 }
10 entity Exploration { source twist as Twist; }
11 entity ModeSelector { source mode as RobotMode; }
12 context ObstacleDetection as Obstacle {
13   source ranges from LaserScan;
14 }
15 context ObstacleZone as Boolean {
16   context ObstacleDetection;
17 }
18 context RandomMotion as Twist {
19   context ObstacleDetection;
20 }
21 context Motion as Twist {
22   source mode from ModeSelector;
23   context RandomMotion;
24   source twist from Exploration;
25 }
26 controller MotionController {
27   context Motion;
28   action TurnAndRoll on Wheel;
29 }

```

Listing 1. An extract of the description of the robotics application with the DiaSpec design language

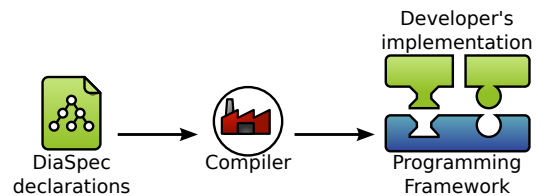


Fig. 2. Overview of the DiaSpec development process

a new entity class, a new context operator, and a new control operator. For this application, we decide to reuse the Twist data type of the ROS middleware which Listing 1 illustrates in line 1.

In this section we saw how to design a robotics application using the SCC architectural pattern and the DiaSpec design language. Both the pattern and the language help decomposing an application in well defined components. Both make it easy to reuse as much as possible from the underlying middleware and existing applications. In the next section we discuss how to implement a robotics application with our approach.

III. IMPLEMENTING A ROBOTICS APPLICATION

The DiaSpec compiler generates a programming framework with respect to a set of declarations for entity classes, context operators and control operators (Figure 2). For each component declaration (entity or operator) the compiler generates an abstract class. The abstract methods in this class represent code to be provided by the developer (*hole* in Figure 2), to allow him to program the application logic (e.g., to trigger an entity action) (*bump* in Figure 2).

Implementing a DiaSpec component is done by *subclassing* the corresponding generated abstract class. In doing so, the developer is required to implement each abstract

method. The developer writes the application code in subclasses, not in the generated abstract classes. This strategy contrasts with generating incomplete source code to be filled by the developer. As a result, in our approach, one can change the DiaSpec declarations and generate a new programming framework without overriding the developer's code. The mismatches between the existing code and the new programming framework are revealed by the Java compiler. To facilitate the implementation process, most Java IDEs are capable of generating class templates based on super abstract classes.

In this section, we give an overview of how to implement some parts of the case study. For a more detailed description, we refer to our previous works [5]–[7].

A. Implementing an operator

For each context or control operator a dedicated abstract class is generated in the programming framework. For each input source of this operator the generated abstract class contains an *abstract method* and a corresponding *calling method*. The abstract method is to be implemented by the developer while the calling method is used by the framework to call the implementation of the abstract method with the expected arguments.

Listing 2 presents a possible Java implementation of the RandomMotion context operator. The onObstacleDetection method is declared abstract in the AbstractRandomMotion generated super class.

Because an operator only manipulates input sources to produce a result, its implementation is independent of any robotics software framework. This facilitates operator reuse for different applications and robots.

B. Implementing an entity

Contrary to operators which are dedicated to the application logic, an entity is at the border between the application and its environment (e.g., the middleware and robot hardware). Implementing an entity thus requires some knowledge of the underlying middleware or hardware.

Listing 3 presents a possible Java implementation of the LaserScan entity class for the ROS middleware. When the middleware publishes a new laser scan message, this message is automatically received by the RosLaserScan instance through the ROS MessageListener interface.

Listing 4 presents a possible Java implementation of the Light entity class for the ROS middleware. The constructor receives a ROS publisher as a parameter which allows the entity implementation to send commands to the robot through the middleware.

C. Deploying an application

Deploying an application requires writing a deployment script in Java. To do this, a developer creates a new Java class by sub-classing the abstract class MainDeploy generated in the programming framework. By doing so the developer is required to implement one abstract method per component and to call the deployAll() method to trigger the deployment. The ROS middleware requires an implementation

```
// Implementation of RandomMotion from Listing 1 line 18
public class RandomMotion extends AbstractRandomMotion {

    // automatically called by the programming framework
    // when ObstacleDetection sends a new value. The method
    // parameter is the value sent by ObstacleDetection
    // whose structure is defined in Listing 1 line 2
    @Override // from super class
    public Twist onObstacleDetection(Obstacle obstacle) {
        Twist cmd = new Twist();
        if (obstacle.getIsDetected())
            // turn
            cmd.angular.z = angleVelocity(obstacle.getRanges());
        else
            // go straight
            cmd.linear.x = new Float(1.0);

        // value transmitted automatically by the programming
        // framework to subscribed operators (here 'Motion')
        return cmd;
    }

    private Float angleVelocity(List<Float> ranges) {
        double left = 0, right = 0;
        // we look to the left and to the right and decide
        // which side has more space
        for (int i = 0; i < middle(ranges); i++)
            left += ranges.get(i);
        for (int i = middle(ranges); i < ranges.size(); i++)
            right += ranges.get(i);
        if (left > right)
            return new Float(-1.0);
        else
            return new Float(1.0);
    }

    private int middle(List<Float> ranges) {
        return ranges.size() / 2;
    }
}
```

Listing 2. A developer-supplied Java implementation of the RandomMotion context operator described in Listing 1. The AbstractRandomMotion super class is automatically generated into the programming framework

```
// Implementation of LaserScan from Listing 1 line 7
public class RosLaserScan extends AbstractLaserScan
implements MessageListener<LaserScan> {

    // triggered when ROS publishes a LaserScan message
    @Override // from ROS MessageListener
    public void onNewMessage(LaserScan message) {
        float[] ranges = message.ranges;
        // sends the list of floats to subscribed
        // context operators through the source defined
        // in Listing 1 line 8
        publishRanges(convert(ranges));
    }

    private List<Float> convert(float[] ranges) {
        // converts a float[] to a List<Float>
    }
}
```

Listing 3. A developer-supplied Java implementation of the LaserScan entity class described in Listing 1, line 7. The AbstractLaserScan super class is automatically generated into the programming framework

```

// Implementation of Light from Listing 1 line 6
public class RosLight extends AbstractLight {

    // A ROS publisher to communicate with the robot
    private final Publisher<Bool> publisher;

    public RosLight(Publisher<Bool> publisher) {
        this.publisher = publisher;
        publish(false);
    }

    // required by design in Listing 1 line 5 and line 6
    @Override // from super class
    protected void on() throws Exception {
        publish(true);
    }

    // required by design in Listing 1 line 5 and line 6
    @Override // from super class
    protected void off() throws Exception {
        publish(false);
    }

    // turns on or off the light depending on the parameter
    private void publish(boolean val) {
        // converts the Java type boolean to the ROS type Bool
        Bool bool = new Bool();
        bool.data = val;
        // asks the robot to trigger its light projector
        publisher.publish(bool);
    }
}

```

Listing 4. A developer-supplied Java implementation of the Light entity class described in Listing 1, line 6. The AbstractLight super class is automatically generated into the programming framework

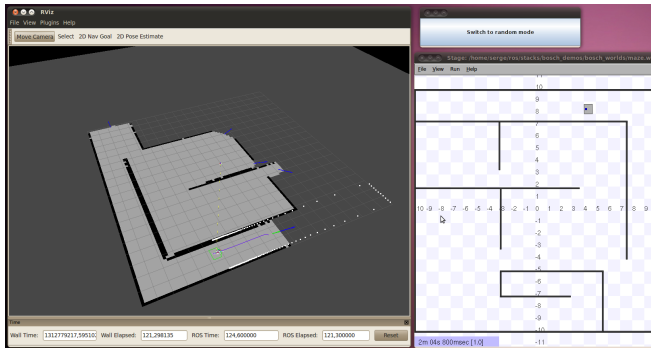


Fig. 3. Screenshot of a simulation of the case study. On the left, a window displays the standard rviz visualization tool presenting the neighborhood visited by the robot in exploration mode. On the top-right, a button allows an operator to change the current mode of the robot. On the bottom-right, a window displays an instance of the Stage simulation engine

of the NodeMain interface. An extract of the deployment script for the case study application is shown in Listing 5.

Figure 3 presents a running simulation of our case study. The code generated is integrated in the ROS middleware and the execution can be analyzed by the tools provided by ROS.

In this section we saw how to implement a robotics application on top of a programming framework generated by the DiaSpec compiler. This programming framework calls developer's code when necessary and make the development easy by passing everything the developer needs as a parameter to abstract methods. In the next section we discuss the benefits and problems of using DiaSpec in a robotics setting.

```

// Deployment script that creates ROS nodes and DiaSpec
// component instances
public class Deploy extends MainDeploy
    implements NodeMain {

    private Node node;

    // starting point defined by ROS
    @Override // from ROS NodeMain
    public void main(NodeConfiguration configuration) {
        // creates a ROS node
        NodeFactory factory = new DefaultNodeFactory();
        node = factory.newNode("laser_cmd", configuration);
        // this is defined in the MainDeploy abstract class,
        // calls all deploy methods
        deployAll();
    }

    // automatically called by the programming framework
    @Override // from super class
    protected void deployRandomMotions(
        Adder<AbstractRandomMotion> adder) {
        // creates a new instance from class in Listing 2
        // and schedules for deployment
        adder.deploy(new RandomMotion());
    }

    // automatically called by the programming framework
    @Override // from super class
    protected void deployLaserScans(
        Adder<AbstractLaserScan> adder) {
        // creates a new instance from class in Listing 3
        RosLaserScan scan = new RosLaserScan();
        // asks ROS to send laser scan messages to scan
        node.newSubscriber("/ATRV/Sick",
            "sensor_msgs/LaserScan", scan);
        // schedules for deployment
        adder.deploy(scan);
    }

    // automatically called by the programming framework
    @Override // from super class
    protected void deployLights(
        Adder<AbstractLight> adder) {
        // allows the application to send messages to ROS
        Publisher<Bool> rosPublisher;
        rosPublisher = node.newPublisher("/ATRV/LightAct",
            "std_msgs/Bool");
        // creates a new instance from class in Listing 4
        RosLight lightPublisher = new RosLight(rosPublisher);
        // schedules for deployment
        adder.deploy(lightPublisher);
    }
}

```

Listing 5. An extract of a developer-supplied Java deployment script for the case-study application

IV. DISCUSSING

DiaSpec decomposes the development of an application into two well defined stages: a design stage for which DiaSpec provides a domain-specific design language and an implementation stage for which DiaSpec provides a design-specific programming framework. With the design language and SCC architectural pattern, a developer is guided in creating components with a single responsibility each, thus enhancing reuse. An application design also explicit interactions between components making the runtime behavior easier to understand. With the programming framework dedicated to the design, a developer is guided in creating an implementation for each component. Indeed, the generated programming framework takes care of the control loop of the application as well as all interactions between the components. As a result, a developer can focus on implementing

the high-level application logic, letting the framework handle the details. Moreover, the programming framework provides all necessary pieces of required information directly as parameters to the abstract methods. This reduces the amount of documentation required to start using the programming framework.

In the previous sections we saw that DiaSpec can be used to design, implement and deploy a robotics application for a widely used middleware. In the following we discuss various problems we have met while applying DiaSpec in a robotics setting.

A. DiaSpec Dynamicity

DiaSpec is capable of handling appearing and disappearing entities at runtime. For example the following code lets the `Motion` context operator subscribe to sources of information from the `Exploration` and `ModeSelector` entities:

```
@Override
protected void postInitialize() {
    discoverExplorationForSubscribe.all().subscribeTwist();
    discoverModeSelectorForSubscribe.all().subscribeMode();
}
```

This method has to be implemented in the `Motion` Java class (for the `Motion` context operator). The programming framework takes care of updating the subscription when a new entity appears or an existing entity becomes inaccessible. As a result, a new exploration mode or a new mode selector can be deployed at runtime. We believe that in a robotics settings where most, if not all, entities are known at deployment time this additional code is most of the time unnecessary. Indeed, this code could potentially be inferred automatically from the declaration of the `Motion` context operator and pushed inside the generated programming framework. However, in a multi-robots settings, where a robot can discover services provided by nearby robots, the DiaSpec entity discovery and subscription mechanisms could still be useful. The DiaSpec design language could be extended to let a developer declare which entities are known at deployment time and which ones should be discovered at runtime. The compiler could then leverage this additional information to generate the necessary code in the programming framework thus reducing the work required by the developers. We plan to investigate this issue in future works.

B. Data Type Reuse

DiaSpec allows the definition of new types (structures and enumerations) as well as the importation of existing Java types. Very often, a middleware such as ROS comes with its own data types. The developer must then choose to reuse the data types coming from the middleware or define new ones. Using the middleware data types can be particularly useful as these data types can be complex such as the ROS “twist” data type. This is the solution we use for the case study and the `Twist` data type as is illustrated by the use of the `import` keyword in Listing 1, line 1. However, choosing reuse of data types from a middleware tightly couples the application with this middleware and thus prevents potential for reuse

of this application with other middleware. Another solution is to develop new data types in DiaSpec. This makes the application independent from any underlying middleware. However, this requires conversion code at the boundaries of the application where communication with the middleware is required. For example, it is possible to define the `Twist` data type within the design as follows:

```
structure Vector3 { x as Float; y as Float; z as Float; }
structure Twist { linear as Vector3; angular as Vector3; }
```

Then, an implementation of the `Wheel` entity would have to convert from the DiaSpec `Twist` type to the ROS `Twist` type:

```
private org.ros.message.geometry_msgs.Twist
    convert(Twist twist) {
    org.ros.message.geometry_msgs.Twist rosTwist;
    rosTwist = new org.ros.message.geometry_msgs.Twist();
    rosTwist.angular = convert(twist.getAngular());
    rosTwist.linear = convert(twist.getLinear());
    return rosTwist;
}
private org.ros.message.geometry_msgs.Vector3
    convert(Vector3 vector) {...}
```

This solution makes the code harder to read and maintain. Moreover, similar code has to be duplicated everywhere in the application where a conversion is required. An intermediate solution is to develop new data types in Java. This solution can embed required conversions in the data type itself to avoid duplication. The resulting code is still harder to read than the first one however.

C. Decomposition grain

During the development of the case study we noticed that following the SCC architectural pattern and the steps proposed in Section II resulted in fine grained components, promoting reuse. It is however important that the developer pays attention not to create too fine grained components which would make the runtime behavior hard to understand and debug. Indeed, because the generated programming framework handles the interactions between the components, debugging very fine grained components requires stepping often into the generated programming framework. This is cumbersome and should not be needed. A possible addition to DiaSpec could involve a dedicated debugger which would let the developer debug his application without stepping into the generated programming framework.

Even with a dedicated development environment, too fine grained components make the system harder to understand. As a rule of thumb, a developer can start by creating a coarse grained component and can refine it when its implementation becomes complex, when the component requires a lot of interactions with other components, or when parts of its computation can be reused.

V. RELATED WORK

Several software engineering approaches have been proposed to lower the complexity of robotics systems [3].

Middleware and Software Frameworks: Numerous middleware and software frameworks have been proposed to support the implementation of robotics applications (e.g., CLARATy [8], ROS [13] and Player/Stage [9]). Such approaches attempt to cover as much of the robotics domain as possible in a single programming framework. This strategy often leads to large APIs, providing little guidance to the developer and requiring boilerplate code to customize the programming framework to the characteristics of the application. In contrast, a DiaSpec-generated programming framework specifically targets one application, limiting the API to methods of interest to the developers. Our code generator could potentially target these middleware thus leveraging existing work and hiding their intricacies from the developer.

Component-Based and Model-Driven Software Engineering: Component-Based Software Engineering for robotics (e.g., [4]) and Model-Driven Engineering for robotics (e.g., OMG RTC [11], SmartSoft [14]) relies on general-purpose notations such as UML to model domain-specific concerns. By using general-purpose and established notations, these approaches leverage existing knowledge from developers and existing tools. Even though such approaches propose a conceptual framework for developing robotics applications, they only provide the user with generic tools. For example, these approaches require developers to directly manipulate UML diagrams, which become “enormous, ambiguous and unwieldy” [12]. In contrast, DiaSpec abstracts away such technologies, limiting the amount of expertise required from the developers.

Domain-Specific Languages: *Smach* is a Python embedded DSL based on hierarchical concurrent state machines for building complex robot behavior from primitive ones [1]. *Smach* is tightly coupled with ROS, allows only static compositions of behavior and can not adapt compositions to new situations during execution. SmartTCL (Smart Task Coordination Language) is an extension of Common Lisp that is used to do on line dynamic reconfiguration of the software components involved in a robot [16]: knowledge bases, simulation engines, symbolic task planners, models and low-level hardware. At design time, the developer defines execution variants that robot operates at runtime. In order to lower robotics inherent complexity, analysis and simulation tools could also be used at runtime to determine pending execution steps with specific parametrisation before the robot effectively execute them. Unlike these DSLs, DiaSpec allows a natural decomposition of applications according to the SCC architectural pattern, guiding the work of the developer. Compared to SmartTCL DiaSpec lacks the ability to recompose the components at runtime.

VI. CONCLUSIONS AND FUTURE WORKS

In this paper, we have proposed to use DiaSpec, a domain-specific design language for Sense/Compute/Control applications, in a robotics setting. We have shown how this language allows a developer to structure an application in fine-grained and reusable components by following the SCC architectural

pattern. Developing a complex application shows the benefits of such an approach regarding reuse of existing software components and diminution of complexity for the developer. We have also highlighted problems we have met during the development of a standard robotics application.

Being able to adapt a robotics system to different capabilities and resources is a key issue in software engineering for robotics. For example, a robot can perform two similar missions differently with different resources. Our approach facilitates changes to a robotics system by making explicit the software components and their interactions. However supporting static adaptation is insufficient in a robotics setting as robots need to dynamically adapt to resource evolutions (e.g., failures and environment) while performing their tasks. *Resource-adaptive architectures* address dynamic adaptations. However, such architectures are ad hoc solutions that developers can hardly reuse and scale. Therefore, an ideal robot control architecture should be *resource-adapting*, i.e., an architecture that explicitly manages and represents resources [2]. The main perspective of this work is to introduce such dynamic variability inside DiaSpec.

VII. ACKNOWLEDGMENTS

The authors gratefully acknowledge the INRIA Phoenix research group who granted the authors the authorization of use of DiaSuite, a tool suite which includes DiaSpec.

REFERENCES

- [1] J. Boren and S. Cousins. The SMACH high-level executive [ROS News]. *Robotics & Automation Magazine*, 17(4):18–20, 2010. doi: 10.1109/MRA.2010.938836.
- [2] Noury Bouraqadi and Serge Stinckwich. Towards an adaptive robot control architecture. In *CAR'2007: Proceedings of the 2nd National Workshop on Control Architectures of Robots, May-June*, pages 135–149, May 2007.
- [3] Davide Brugali, editor. *Software Engineering for Experimental Robotics*. (Springer Tracts in Advanced Robotics). Springer, March 2007.
- [4] Davide Brugali, Alex Brooks, Anthony Cowley, Carle Côté, Antonio Domínguez-Brito, Dominic Létourneau, Francis Michaud, and Christian Schlegel. Trends in component-based robotics. In Davide Brugali, editor, *Software Engineering for Experimental Robotics*, volume 30 of *Springer Tracts in Advanced Robotics*, chapter 8, pages 135–142. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. doi: 10.1007/978-3-540-68951-5_8.
- [5] Damien Cassou. *Développement logiciel orienté paradigme de conception : la programmation dirigée par la spécification*. PhD thesis, University of Bordeaux, March 2011. (in French). Available from: <http://tel.archives-ouvertes.fr/docs/00/59/03/61/PDF/thesis.pdf>.
- [6] Damien Cassou, Emilie Balland, Charles Consel, and Julia Lawall. Leveraging software architectures to guide and verify the development of sense/compute/control applications. In *ICSE'11: Proceedings of the 33rd International Conference on Software Engineering*, pages 431–440, Honolulu, HI, USA, May 2011. ACM. Available from: <http://hal.inria.fr/docs/00/53/77/89/PDF/icse2011.pdf>, doi:10.1145/1985793.1985852.
- [7] Damien Cassou, Benjamin Bertran, Nicolas Lorient, and Charles Consel. A generative programming approach to developing pervasive computing systems. In *GPCE'09: Proceedings of the 8th International Conference on Generative Programming and Component Engineering*, pages 137–146, Denver, CO, USA, October 2009. ACM. Available from: <http://hal.inria.fr/inria-00405819/PDF/gpce42-cassou.pdf>, doi:10.1145/1621607.1621629.
- [8] CLARATy: robotic software framework. <http://claraty.jpl.nasa.gov>. Available from: <http://claraty.jpl.nasa.gov>.

- [9] Toby H.J. Collett, Bruce A. MacDonald, and Brian P. Gerkey. Player 2.0: Toward a practical robot programming framework. In *ACRA'05: Proceedings of the 7th Australasian Conference on Robotics and Automation*, pages 1–9, Sydney, Australia, 2005. ARAA.
- [10] George Edwards, Joshua Garcia, Hossein Tajalli, Daniel Popescu, Nenad Medvidovic, Gaurav Sukhatme, and Brad Petrus. Architecture-driven self-adaptation and self-management in robotics systems. In *SEAMS'09: Proceedings of the 4th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 142–151, Los Alamitos, CA, USA, May 2009. IEEE Computer Society. doi:10.1109/SEAMS.2009.5069083.
- [11] Omg rtc: Robotic technology component specification 1.0, 2008. Available from: <http://www.omg.org/spec/RTC>.
- [12] Ruben Picck and Vjeran Strahonja. Model Driven Development - future or failure of software development? In *IIS'07: Proceedings of the 18th International Conference on Information and Intelligent Systems*, pages 407–413, Varazdin, Croatia, September 2007. Aurer, Boris.
- [13] ROS: Robot Operating System. <http://www.ros.org>. Available from: <http://www.ros.org>.
- [14] Christian Schlegel, Thomas Haßler, Alex Lotz, and Andreas Steck. Robotic software systems: From code-driven to model-driven designs. In *ICAR'09: Proceedings of the 14th International Conference on Advanced Robotics*, pages 1–8, Munich, Germany, June 2009.
- [15] Bruno Siciliano and Oussama Khatib, editors. *Springer Handbook of Robotics*. Springer, Berlin, Heidelberg, 2008. doi:10.1007/978-3-540-30301-5.
- [16] Andreas Steck and Christian Schlegel. Managing execution variants in task coordination by exploiting design-time models at run-time. In *IROS'11: Proceedings of the 24th International Conference on Intelligent Robots and Systems*, San Francisco, CA, USA, September 2011. IEEE. (to appear).
- [17] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, New York, NY, USA, 2009.
- [18] Brian Yamauchi. Frontier-based exploration using multiple robots. In *AGENTS'98: Proceedings of the 2nd International Conference on Autonomous agents*, pages 47–53, Minneapolis, MN, USA, 1998. ACM. doi:10.1145/280765.280773.