Intel Serv Robotics (2014) 7:53–65
DOI 10.1007/s11370-014-0145-y

SPECIAL ISSUE

Dynamic State Charts: composition and coordination of complex robot behavior and reuse of action plots

Dennis Stampfer · Christian Schlegel

Received: 9 August 2013 / Accepted: 27 January 2014 / Published online: 4 March 2014

Abstract Support for separation of roles is decisive towards a successful business ecosystem where various stakeholders with dedicated expertises network and collaborate. However, it depends on means for composition(system of systems, reuse of black boxes). This paper proposes Dynamic State Charts as an extension of state charts for composition and coordination of complex robot behavior which is one of the challenges in service robotics. Their states allow to refine their content by choosing from a set of alternative matching state instances for robust task execution and to manage the complexity of real-world tasks. Dynamic State Charts allow reuse and can be bundled with software components and are provided in a repository (idea of an "robot app store") as a step towards composition and separation of roles as necessary for a business ecosystem in service robotics. The approach is demonstrated in a practical application with a service robot.

Keywords Service robotics · Task coordination · Model-driven software development · Robotic software development · State charts

1 Introduction

Impressive algorithms and abilities for service robots have been developed in recent years and the challenge is to integrate them. This requires mechanisms for their control and coordination to perform tasks. Tasks should be composable to more complex tasks.

Technically, a task (e.g., grasp cup) requires knowledge about the coordination of the abilities of a robot, e.g., parametrization of components (e.g., object recognition, manipulation planning), further subtasks (first determine object pose, then grasp it, ...) and handling of deviations in executing the task (e.g., object slipped out of gripper). A sequence of such tasks is called an action plot. An expert in a particular application domain (task expert) should be able to encode new action plots in his domain by (re)using provided action plots from other experts but should not need to know about their internal details. This is necessary to separate responsibilities and roles.

Hierarchical state charts [8] are adequate for robot behavior coordination in many applications. Entry- and exit-actions can reconfigure system components (parameters, data flow, control flow) according to the progress of task execution as reported by events. State charts are convenient due to their hierarchies, parallelism, event-driven nature, defined semantics and existing (graphical) tools. Nevertheless, state charts so far lack a mechanism for instantiation: if an action plot described as state chart (e.g., determine the pose of an object) is needed within a different more complex action plot (e.g., approach shelf and fetch object, approach table and fetch object), it needs to be copy-pasted as nested state chart (including code adaptions and parameterization) into all action plots using it. This makes their reuse nearly impossible.

Reuse of algorithms and components that wrap the robot's abilities (and algorithms) is state of the art [28,31]. Even more, task coordination is an active research field. However, task coordination (action plots) must also be reusable and get more attention in the overall development process: i.e., reuse of the description and method for coordinating components

D. Stampfer (✉) · C. Schlegel
Department of Computer Science, University of Applied Sciences Ulm, Prittwitzstr. 10, 89075 Ulm, Germany
e-mail: stampfer@hs-ulm.de
URL: http://www.servicerobotik-ulm.de

C. Schlegel
e-mail: schlegel@hs-ulm.de

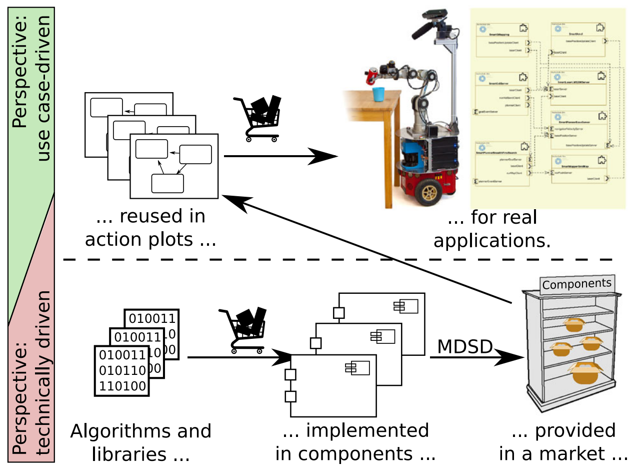

Perspective: use case-driven
Perspective: technically driven
... reused in action plots ...
... for real applications.
Algorithms and libraries ...
... implemented in components ...
MDSD
Components
... provided in a market ...

Fig. 1 Until now, the development of robotic applications is mostly technically driven and must be extended with a more use-case driven perspective. We exploit model-driven software development (MDSD) for separation of roles and support of different perspectives

to perform complex tasks. This enables the separation of different roles that are involved in the development process.

To achieve this separation, according views must be established for every role. The available views so far were foremost technically driven and related to specific problem solutions. To get matured applications that “do” something useful to a customer or user, the use-case view becomes more important (Fig. 1). It is therefore necessary to complement the technical perspective at some point in the development process by a more use-case driven perspective. Having reusable software components and action plots available as building blocks, allows to focus on the use case rather than technical details. Model-driven software development (MDSD) can be exploited to realize this.

All stakeholders can benefit from existing and matured building blocks where reuse improves quality. They do not any longer need to be an expert in every field and can focus on their contributions. This speeds up the development process and decreases time-to-market. At the same time, it increases return-on-investment by amortization of development costs by multiple reuse. Finally, this enables evolvement of a market for building blocks.

We propose Dynamic State Charts which support instantiation of dynamic states as a special kind of a state [32]. They extend regular event-driven state charts, most importantly with dynamic states which only consists of the hull of a state. Alternatives and variations can be defined at design time but its internals are selected from a set of matching instantiations from a library at runtime. Dynamic State Charts support reuse of action plots as building blocks for composing complex behaviors and publishing them following the idea of the “robot app store”. They increase maintainability through onetime representation of repetitive state chart blocks and support robustness by context and situation dependent selection of alternative instantiations of states (e.g., following a person based on a laser ranger or following a person based on vision). They further reuse the knowledge of how to perform complete tasks independently of the refinements: for example, a state chart describing the action plot of cleaning up the table is independent from which kind of object recognition is being used in the sub-steps.

The foundation of this approach is based on insights and experience gained by implementing and operating a variety of real-world and complex robotics applications like e.g., the robot butler scenario [22]. This article presents an extension of [33].

2 Related work

To deal with the growing complexity of robotic systems, the robotics community has established concepts to improve robustness, interoperability, maintainability and reusability by means of component-based architectures and model-driven software development. This resulted in robotic frameworks and architectures such as ROS [20], SmartSoft [24,31] and Orocos [2].

Current activities focus on composition towards reuse as black boxes and configuration at runtime for both, parameters and the component’s lifecycle. The BRICS component model (BCM) [3] focuses on separation of concerns. It introduces the concept of composition with components grouped or nested together. They include a lifecycle coordinator to form a new reusable component (e.g., components for localization, mapping, path planning as a navigation component).

Restricted finite state machines (rFSM) [12] were developed in the context of BRICS/BCM. rFSM are a minimal variant of Harel state charts and integrate into the robotic framework OROCOS/RTT. It was developed with a focus on component coordination for robotics. It is not intended to coordinate a whole system of components in the sense of a sequencer since it misses a generic component configuration interface.

BCM and rFSM make valuable progress towards reuse and composition of components. However, coordination is too closely linked to the component level. rFSM Coordinators inside composite components control the component’s lifecycle rather than the component’s skills at task level (in our approach, such lifecycle automatons are a part of every component [25] and are used by but not for task coordination). Therefore, their coordination is technically driven rather than use case driven. Furthermore, rFSM does not support dynamic states as introduced in this paper.

Even though the separation of concerns in component-based architectures (e.g., as in BCM) is a valuable step towards real applications, we believe that the separation in this functional part is insufficient: separation must be applied at all levels and to all stakeholders in the development process. SmartSoft and SmartMDSD already support separation of roles as needed in a robotics business ecosystem [26].

In this article, we go a step further in reuse and composition of parts related to computation and functionality (components). We make reuse at a task level / use-case level: reuse and composition of action plots.

With respect to robot control, first developments resulted in languages like the reactive action packages (RAPs) [5]. RAPs implement situation-driven execution and expand a sketchy plan to concrete actions at runtime.

State charts [8] are since their invention accepted and they are in widespread use in computer science. Even though state charts have been used for robot applications [7,10], only few approaches such as XABSL, SHSM and SMACH use them for generic robot control.

The extensible agent behavior specification language (XABSL) [21] supports hierarchical decomposition of behavior and provides mechanisms for action selection with finite state machines. It is well suited for highly reactive applications such as RoboCup soccer. The execution mechanism is based on the cyclic traversal of finite state machines expressed as decision trees. However, it does not follow the idea of event-based systems and situation-driven execution.

Skill hybrid state machine (SHSM) [15] is a behavior engine implemented in the Lua Programming Language [19]. It was mainly developed for the Humanoid Robot Nao and RoboCup. It allows hierarchical composition of hybrid state machines with two final states success and failure. However, the restriction to these two final states is limiting (cf. [6]). The decision about success or failure of a task must be the decision of the caller as only it has the knowledge about the context in which the call has been issued. It is therefore the only one that can decide whether the result of an action is to be interpreted as success or failure. SHSM is also used to coordinate the processing steps of algorithms. However, this approach is too low level for task control.

The state machine executive (SMACH) [1] has successfully been used for scenarios like fetching beer or playing pool. It defines hierarchies of state machines that return outcomes (user-defined strings) to proceed to the next state. Such a transition is thus a mapping of one return string value to another state and can only be triggered internally but not from the calling state. SMACH supports data passing between state hierarchies via explicitly defined user-data structures. It explicitly uses blocking calls which may cause the complete execution system to not respond anymore. SMACH states "correspond more to states in structured programming" [23]. It is thus essentially only a flowchart since it proceeds from one activity to the next upon completion based on input of the states itself. It is not event based and does not react to external events. It is thus a smart structuring of function calls aimed at rapid prototyping of scenarios [1]. SMACH supports static composition of behaviors without the ability of situation-dependent task execution.

Only few approaches separate task modeling from reactive components. For example, SmartTCL [34] is a domain-specific language in LISP for task coordination. It uses the same coordination services of components as Dynamic State Charts. It supports hierarchical task decomposition and situation-driven execution of tasks. Task-nets are specified at design-time with variation points purposefully left open. Task-trees are at runtime dynamically created and modified. SmartTCL focuses on the integration of external planners for specific problem domains (e.g., Metric-FF) and reasoning.

Simplicity and clarity of the languages is the common goal of the presented state-chart-like approaches. Hierarchies of actions are a key mechanism in all presented approaches. Hierarchies require to identify the proper abstraction level below which low-level processing apart from state charts takes place (SMACH states are also intended to compute results [23], SHSM have algorithms in states). Most approaches lack mechanisms for generic configuration of software components in a robotic system. The focus on reuse, if any, is on "local" reuse within the application but not towards composing new applications at the task modeling level in a building blocks manner. Only few approaches are event based and thus not as responsive as required for task coordination: e.g., SMACH and SHSM allow no transitions into higher levels than the current state. Many do not address the runtime refinement and situation-dependent selection of instantiations of states.

3 Use case

Nowadays, the roles in developing robots are too tightly coupled and every involved person needs to be an expert in every area ranging from algorithms over expertise of the application domain up to the final integration. This led to impressive demonstrations of robot skills in laboratories, but these have been mainly driven by technical achievements. A robotics expert can neither be an expert in each of the prospective application domains nor can a robotics company become a player in all the markets of these application domains.

The lack of separation of roles is a severe show-stopper when it comes to developing a service robotics market [26]. Separation of roles reduces risks, efforts and costs as well as time-to-market and increases overall robustness of systems. To exploit economically viable service robotic applications, we need to make a shift towards use-case driven service robot systems. This requires the involvement and collaboration of stakeholders with dedicated expertises as shown in Fig. 2, ranging from component developers over system integrators

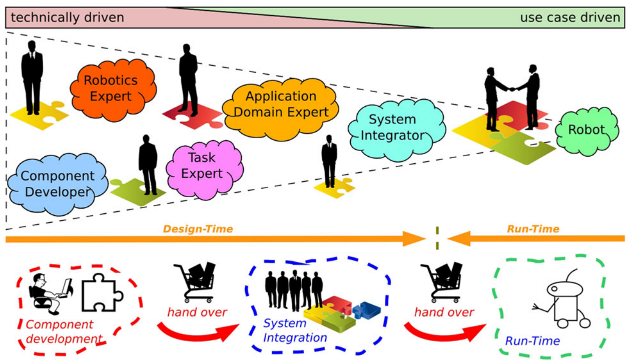


Fig. 2 Separation of roles in a robotics business ecosystem to make the step from technically driven service robot systems towards use-case driven ones. At a technical level, it requires handover of composable black-boxes

and application domain experts to the robot itself that might extend its capabilities at runtime by downloads via the internet.

A successful business ecosystem depends on separation of roles where building blocks can be handed over as black-box from one role to another, hiding complexity and still ensuring composability.

To realize this separation, we have to follow a development process including appropriate tooling (like model-driven approaches) which provide partial but focused views on the overall system development. These views have to be aligned with the needs and skills of the different stakeholders: the architectural view needs to be separated from the implementation as well as the technical views need to be separated from the business logic / use case. This not only decouples development in space but also in time which enables speedup of the overall development and faster time-to-market.

An abstract use case with respect to task coordination is illustrated in Fig. 3.

Component developers model the component hull of a component (Fig. 3: ①) with provided/required services. They are based on communication patterns [27] and provide stable interfaces towards the outside view of a component (to be compatible with other components) as well as to the inside view of a component (for implementing functionality, either by reusing existing libraries or implementing own algorithms, Fig. 3: ②).

In this step, they also define the coordination services visible at the component hull and write minimal action plots that make these components usable in action plot models (Fig. 3: ③). For example, they model components for collision avoidance, mapping and path planning and model simple action plots that can navigate (a “goto” action plot, Fig. 3: ④).

They focus on implementation and component functionality rather than integration. They are interested in keeping effort and costs low by reusing algorithms and libraries and want to offer and sell their work (components) in the market for reuse.

Task experts have deep knowledge in task modeling. They develop abstract action plots (Fig. 3: ④, ⑤) in cooperation with application domain experts. These plots focus on the task at hand which is independent of the specific robot or components used in the final application. For example, they model an action plot for a delivery robot but leave open how the robot moves (walk, drive or fly) or navigates (SLAM or indoor GPS, 2D or 3D path planning) to its destinations since all delivery robots will somehow move and navigate. They are interested in reusing action plots (their own or others) and want to offer and sell their work (action plots) in the market for reuse.

System integrators put together action plots and components (Fig. 3: ⑥) and fit both to the concrete robot and

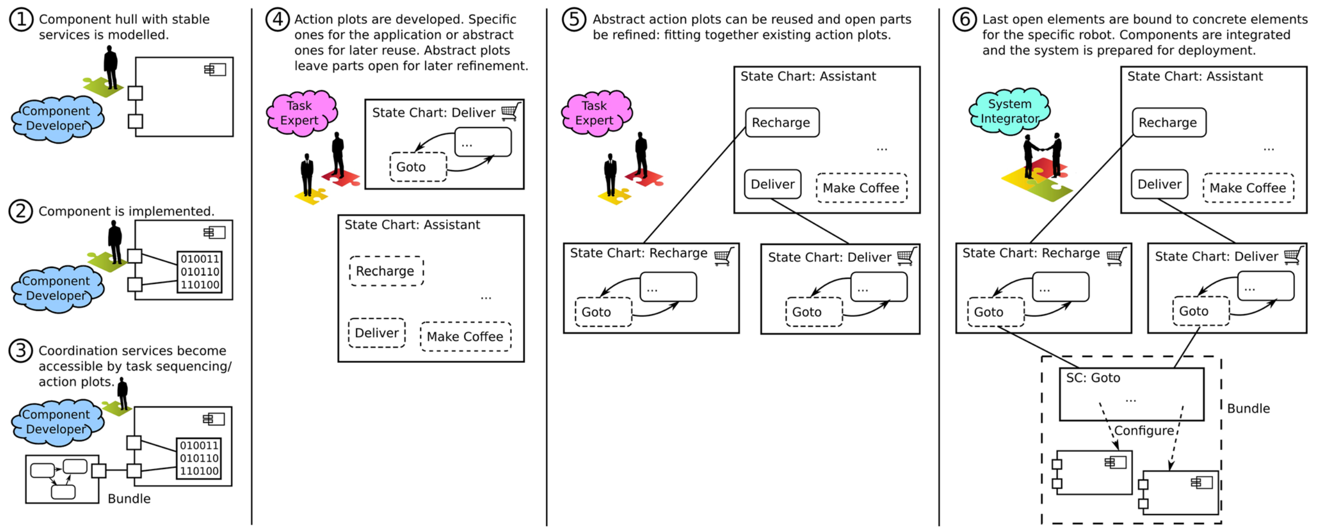

① Component hull with stable services is modelled.
Component Developer
② Component is implemented.
Component Developer
010011
010110
110100
③ Coordination services become accessible by task sequencing/ action plots.
Component Developer
Bundle
④ Action plots are developed. Specific ones for the application or abstract ones for later reuse. Abstract plots leave parts open for later refinement.
Task Expert
State Chart: Deliver
Goto
...
State Chart: Assistant
Recharge
Deliver
Make Coffee
⑤ Abstract action plots can be reused and open parts be refined: fitting together existing action plots.
Task Expert
State Chart: Assistant
Recharge
Deliver
Make Coffee
State Chart: Recharge
State Chart: Deliver
Goto
⑥ Last open elements are bound to concrete elements for the specific robot. Components are integrated and the system is prepared for deployment.
System Integrator
State Chart: Assistant
Recharge
Deliver
Make Coffee
State Chart: Recharge
State Chart: Deliver
Goto
SC: Goto
Configure
Bundle

Fig. 3 Use case for modeling of task coordination from defining a component hull with stable services to a complete action plot for an application

application. Both, action plots and components are treated as building blocks, i.e., they are black boxes. System integrators do not want to be bothered by internals of components or action plots. They want to reduce effort by reusing existing components and action plots.

Application domain experts know best about typical use cases in the target domain of the final robot application. They do not need knowledge of robotics or computer science. For example, a cleaning professional is an expert in cleaning but does not know anything about robotics.

Robotic experts have deep knowledge of robotics and transfer the application domain expert's knowledge towards the robotics application and "build" the robot physically. For example, he knows best what kind of robot (walking, driving, flying) and additional details are best for the application.

The robot executes the action plot as coordinated by the sequencer and thereby orchestrates components to work together as an application. Alternatives and variations in the action plots are chosen from a library at runtime depending on the context and situation (Dynamic states).

4 MDSD for behavior modeling

We are convinced that model-driven software development (MDSD) is an enabler to drive robotics towards a business ecosystem for robotics software [26].

In the context of task modeling and composition, it is important to model independently from the implementation or from deep technical details or even program code, e.g., architecture must be separated from implementation. Due to several involved stakeholders, elements must be handed over from one stakeholder to another and their different views and needs must be supported and provided. While every stakeholder has to be given sufficient degrees of freedom, the boundaries must be clearly defined for a successful handover.

This can be achieved by establishing MDSD throughout the whole development process where it is an important tool implementing the separation of roles, supporting the stakeholder's perspective and reducing time of development.

4.1 Graphical and textual modeling

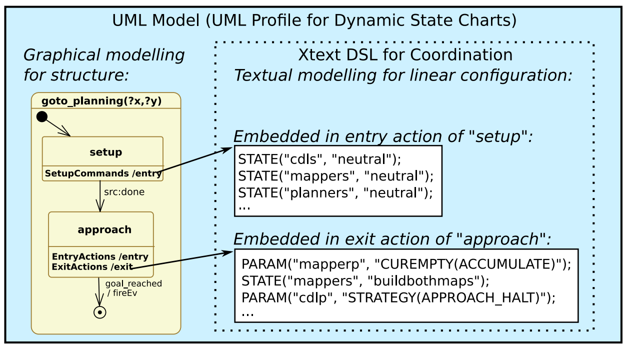
For MDSD, both graphical and textual modeling are used. As there are arguments for both representations, this section elaborates how and why which representation is used.

Graphical modeling (e.g., with Eclipse GMF, UML) is good for visualizing structural relations and getting information out of descriptions at the first glance. This is important for behavior modeling. They are easier to understand, easier to use by (novice) users and may even be used in discussions with non-experts when not all aspects of a diagram are important and therefore helpful for separation of roles.

From the modeling point of view, textual modeling (e.g., Xtext, Eclipse TMF) can be compared to writing code, so its use in domains involving "algorithmic" models is likely. Compared to graphical models, textual models have their advantages when models are "linear", i.e., lists of properties or name value pairs. These kinds of models look the same when they are modeled textually or graphically. Tools for textual models are more matured, lightweight and can easily be used with tools like diff and versioning tools.

A common use case in the modeling world is to combine graphical and textual modeling such that two views exist of a single model: the model can be edited in its graphical and in its textual representation simultaneously as e.g., presented in [14]. However, both representations of the model need to be maintained and additional effort in layouting is required: when the textual model is updated (elements added), the graphical representation must be updated as well and automatic placement of elements is difficult because information is missing (e.g., size, position of the element in a 2D drawing plane). Vice versa, when graphically adding elements at a 2D position, the textual view must be updated, but again, information is missing (e.g., where in the linear list of lines shall the object be added?).

Our approach is to make use of the individual strengths of graphical/textual modeling. We use graphical modeling for elements where structure is important: in the action plot. Since state charts have a graphical representation, we keep it and reuse UML (Papyrus UML [18] modeling from Eclipse). Coordination of components (configurations in entry/exit actions) in lists of values, parameters, etc. corresponds more to linear programming, so we use a textual modeling language (Eclipse TMF/Xtext DSL [4]) for this purpose and embed this language into the entry/exit actions of the UML state chart (Fig. 4). This is similar to YAKINDU SCT Tools [11].


UML Model (UML Profile for Dynamic State Charts)
Graphical modelling for structure:
goto_planning(?x,?y)
setup
SetupCommands /entry
src:done
approach
EntryActions /entry
ExitActions /exit
goal_reached / fireEv
Xtext DSL for Coordination
Textual modelling for linear configuration:
Embedded in entry action of "setup":
Embedded in exit action of "approach":

```
STATE("cdls", "neutral");
STATE("mappers", "neutral");
STATE("planners", "neutral");
...
```

```
PARAM("mapperp", "CUREMPTY(ACCUMULATE)");
STATE("mappers", "buildbothmaps");
PARAM("cdlp", "STRATEGY(APPROACH_HALT)");
...
```

Fig. 4 Use of graphical and textual modeling: the structure of the action plot is modeled graphically using Papyrus [18] and UML profiles. Configuration of components is modeled textually in Xtext [4]

5 Robot behavior with Dynamic State Charts

Dynamic State Charts are based on UML state machines [17] (the term "state machine" is used in the context of "UML::StateMachine"). Our implementation uses UML profiles which allow to make use of existing UML toolchains and the UML standard.

5.1 Architecture: interaction of sequencer and skills

This work is based on a layered architecture as described in [34] with a sequencing layer and a skill layer (Fig. 5). Low-level processing in closely coupled reactive control loops is done in components in the skill layer. These components include algorithms and low-level control accessible as skills. They provide basic services such as collision-free driving and path planning. Skills can be configured at runtime. For example, navigation is done by setting up path planning to get input from map building and send intermediate way-points to the motion execution that avoids obstacles (Fig. 5).

The sequencing layer consists of a central component (sequencer) that coordinates the execution of the action plot encoded in Dynamic State Charts and triggered by reported events. Persistent data to build a world model (e.g., locations of objects) are stored in and retrieved from an external knowledge base. This gives space for powerful reasoning mechanisms.

This architecture is not fixed to two layers or any layers at all. In fact, the sequencer is the central component that orchestrates the others.

The component-based architecture follows the principles of SMARTSOFT and the SMARTMARS component meta

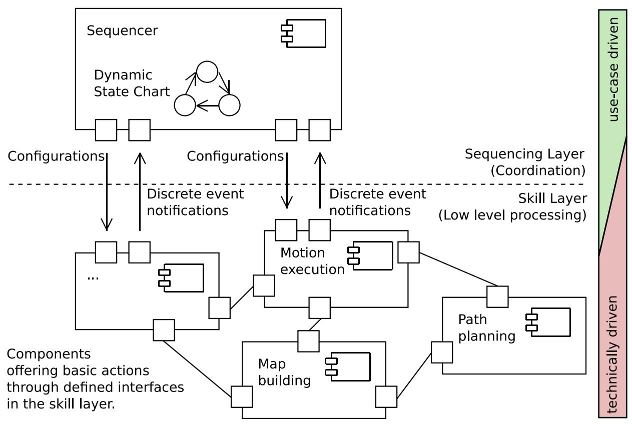

Sequencer
Dynamic State Chart
Configurations
Configurations
Discrete event notifications
Discrete event notifications
Sequencing Layer (Coordination)
Skill Layer (Low level processing)
use-case driven
technically driven
...
Motion execution
Path planning
Map building
Components offering basic actions through defined interfaces in the skill layer.

Fig. 5 An architecture with two layers for Dynamic State Charts. The skill layer consists of components running algorithms and low-level processing in continuously running control loops. These skill components can be configured by the sequencer according to the action plot (e.g., motion execution, path planning and map building to drive to position *X*/*Y*). Skills report about execution via discrete events to the sequencer

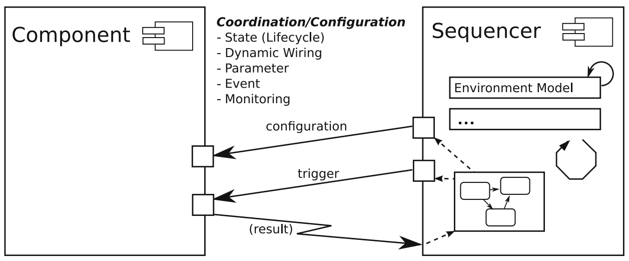


Fig. 6 The sequencer typically coordinates by first configuring components, then triggering them to start an action (both via parameter pattern) and finally can collect results via event pattern

model [28,31] with well-defined services through communication patterns to communicate with components. In general, the sequencer coordinates the components by first configuring them (e.g., configuring objects to recognize), then triggering the component to start (recognize objects) and finally collecting results or feedback. This is done using a set of communication patterns for coordination (orchestration), e.g., state, parameter, event (Fig. 6). The state pattern [25,28] is used to switch between modes of processing in skills (e.g., run/standby). The parameter pattern [28] is used to configure and trigger actions in skills. Skills are expected to report to the sequencer about actions using events (event pattern [24,28], e.g., goal reached). Necessary information is included in the event.

The sequencer coordinates and orchestrates skills by configuring components according to the action plots encoded in Dynamic State Charts. All communication to the sequencing layer is asynchronous to stay responsive through all hierarchies of states in the state chart.

The sequencer only triggers actions and receives discrete event notifications that report on the execution process. This enables task sequencing at a high level of abstraction as there is no data communication on an algorithmic level or even closely coupled control loops in between sequencer and skill components (e.g., visual servoing). By responding to events, the responsibility for the action is on the skill layer and follows the concept of cognizant failure [16]: any problem encountered which cannot be handled locally must be notified to the sequencer. The state chart in the sequencer then takes command and can resolve a problem by reconfiguring the skill layer as defined by the action plot.

5.2 State charts as sequencer

Each state of the Dynamic State Charts resembles a configuration of the skill layer (Fig. 7). Configurations are set up within entry actions and can be reverted by exit actions (i.e., to set in a safe state if necessary). Further hierarchical refinement of the configurations is made along the hierarchy in the entry/exit actions of nested state charts. Transitions are taken

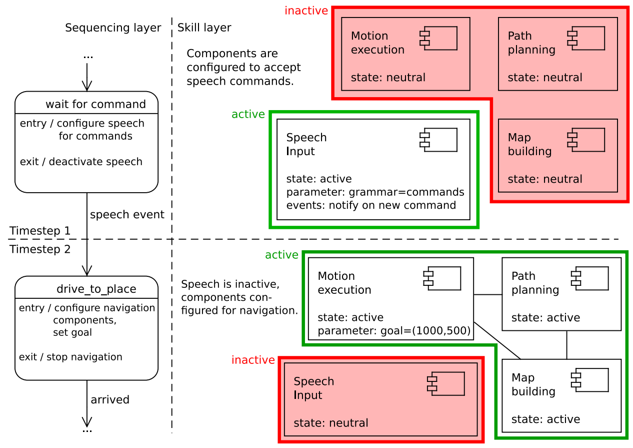


Fig. 7 Each state (*left*) corresponds to a configuration of the skill layer (*right*). The first state in this example (timestep 1) configures skills for speech recognition (and sets navigation components to inactive). After a recognized voice message was sent to the sequencer, the second state (timestep 2) configures the skill layer to approach a location. The entry actions in each state set up the configurations

according to events raised by components. They continue to the next state and reconfigure the components in entry/exit actions (Fig. 7).

Communication with skill components from within the state chart is done using a set of function calls that map communication with the coordination interface of skill components (Fig. 6). A mapping assigns a user-defined name for each component port in the skill layer. Configurations may, for example, include the change of the mode (SmartSoft lifecycle automaton/state pattern [25]) of a skill component. The call from the state chart is translated into communication to the state port of the component.

Communication that the state chart sends or receives towards the framework is buffered in queues (Fig. 10). Fired framework events are pushed to an event queue that is then dispatched to the sequencer. The dispatcher translates these events to framework events. Actions that leave the state chart (for configuring components) are again queued and translated into framework communication for the coordination patterns. The state chart may fire events for itself. They are put in the action queue to meet the run-to-completion and are again forwarded to the event queue by the dispatcher.

The sequencer does not execute any actions or control loops itself but only triggers them in skills—the do action of states is therefore not used. The state chart manages parallel control flows with regions (e.g., driving to location), while the corresponding actions (map building, collision avoidance, motion control, …) are executed by components.

5.3 Dynamic states for situation-driven execution

Dynamic State Charts extend state charts by dynamic states. State charts in general are sufficient for robot control in many

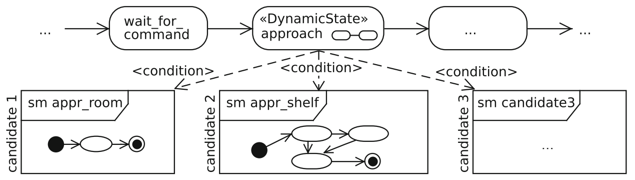


Fig. 8 Dynamic states select their content at runtime from alternative action plots (candidates) based on the current situation. The states from the selected alternative action plot become the substates of the dynamic state

use cases where full explication of all execution variants is feasible beforehand. However, modeling all possible states and transitions that are necessary for real-world tasks including all variants and recovery tasks (including deviations from the regular action plot), soon results in a huge, unreadable and error-prone representation.

Dynamic states are introduced for this purpose. They allow the developer to leave certain states purposefully empty for runtime refinement from candidates. They are essentially submachine states with the referenced state machine being determined depending on the current situation (Fig. 8). The benefit is a simpler and more manageable representation that allows the situated selection of skill configurations. For example, only one dynamic state is used to represent approaching a location (approach). The correct refining (submachine) is chosen at runtime from a library (approach room or approach shelf which may be different approach strategies).

Candidates are specified in a top-down direction (a dynamic statereferences possible candidates). A logic condition for selecting a candidate is defined in the candidate itself (bottom-up direction). Each candidate can thus decide whether it is appropriate in the current situation (same as information expert principle [13]). The condition is not defined in the dynamic state because this resembles a choice pseudo state and leads to redundant conditions.

To express the appropriateness of a candidate in a logical condition, it checks the current situation by evaluating parameters passed to the state or stored in the knowledge base (e.g., the type of approach strategy depends on whether the location is a person, object or room). In case no candidate matches, a special event is fired that can be handled on a transition leaving the dynamic state.

5.4 Using actions for coordination/configuration

Dynamic State Charts distinguish between internal and external actions which improves the structure significantly. Internal actions stay within the state chart (e.g., setting variables) and have no influence on components. They are allowed at transitions.

External actions have effects outside the sequencer towards components. They may be used for configuration through the coordination/configuration service (Fig. 6) for setting parameters, states or requesting and activating events. Actions are modeled in a textual DSL. External actions are only allowed in entry/exit actions of states.

5.5 External events

Events in SMARTSOFT (components) need to be activated [24]. This way, the activator configures in which cases it wants to be notified by events. For example, the activator wishes to be informed as soon as the battery drops below 5%.

States can activate events of components and assign user-defined state chart event names. Event names separate the specific event activation with the exact activation parameters from the reaction to the event. In Fig. 9, this allows the usage of an abstract event label low_bat on a transition without knowing the activation details that caused the event to fire (e.g., battery $<$ 5%). Additional event details may be included in the state chart event attributes (e.g., exact battery level). Component events are translated to these named state chart events when fired (Fig. 10).

Events in the state chart are scoped. By default they are only visible within the state machine they were activated in (to transition between children). Events can be declared to be visible on transitions that leave the state. This way, activated events can be handed over to the next level in the hierarchy to consider it for further transition.

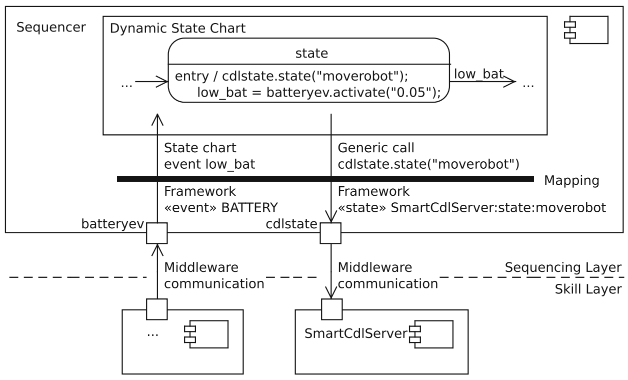


Fig. 9 Mapping between state chart and skills: a set of function calls in the state chart are translated into communication to skills. Component events received from skills are translated into state chart events and fire transitions

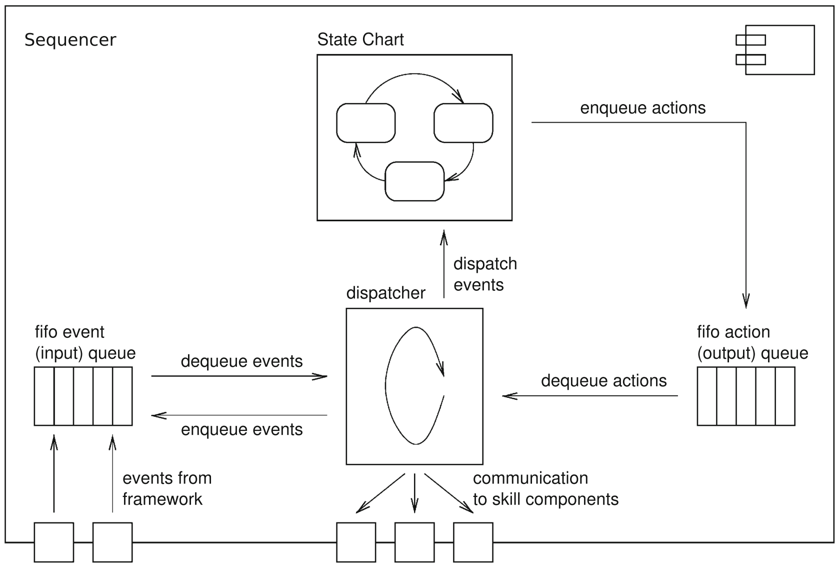
5.6 Data passing and internal events

One-way parameter passing (as defined for state machines in UML [17]) is used to send data into states and can then be read from local variables.

Between states, data are passed with scoped variables (Fig. 11). Variables have a local scope and are visible within the containing state machine (the scope is defined by the border of the state machine).

State charts can fire internal state machine events that carry variables to pass data in attributes to the parent hierarchy and to report on the outcome of actions to consider the next state. Figure 11 shows the usage of variables, parameter passing and events: the event evt is fired by the submachine say and carries attributes that can be accessed. Special success or failure states are not available as the submachine cannot decide on the success or failure.

Fig. 10 Handling of events and actions between component hull and state chart in the sequencer component



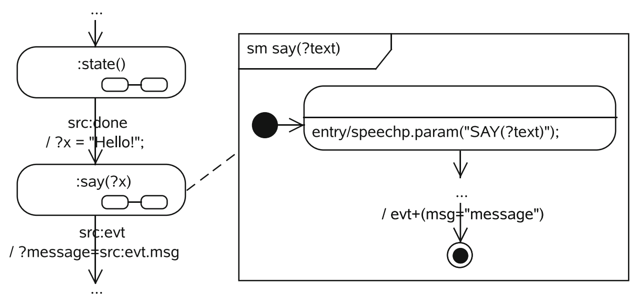


Fig. 11 Assignment and use of variables: ?*x* is passed as parameter into the submachine. Event evt returns data in an attribute which is saved to ?message

5.7 Grouping and hierarchy

In Dynamic State Charts, composite states are only used to structure and group states. They do not introduce a new scope or hierarchy as they bring no further hierarchy. This follows a suggestion by Simons [29] who suggests to use composite states only for abbreviation to draw group transitions only once.

Submachine states are supported to bring hierarchy and abstraction as they reference a state machine that is completely encapsulated.

5.8 Transitions

External transitions as defined e.g., by Harel and UML state charts define transitions that cross the boundary of either composite states or submachine states. They are intended to jump from any state to any other state regardless of the hierarchy or any other limitation, even between states of different state machines. They are not supported in the presented approach as they break encapsulation and have a negative impact on the readability [30].

5.9 On composability and reuse

Dynamic State Charts are encapsulated state machines with a defined interface. Developers can use them as a black-box without knowing about the internals. This raises the abstraction by each hierarchy which helps to manage the complexity of the overall action plot and allows for separation of roles.

With this described concept, two main kinds of Dynamic State Charts can be identified based on what mechanisms are being used (Fig. 12), primitive action plots and abstract action plots.

Primitive action plots contain communication calls to skill components through the coordination/configuration services. They thus build the bridge to skill layer configurations and at the same time from task modeling to component configuration. They contain details for the configuration of individual components (e.g., parameters) and are thus specific to them.

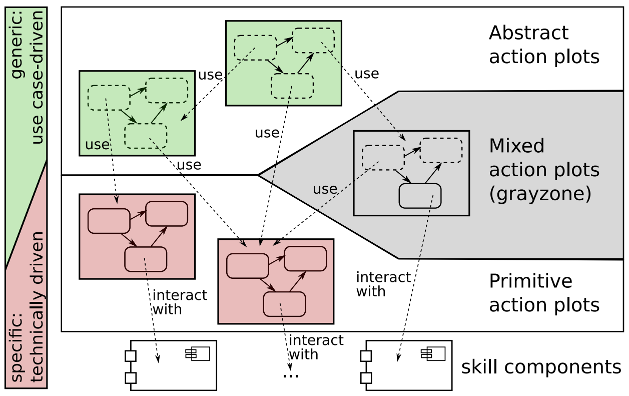


Fig. 12 Action plots can be divided into primitive and abstract action plots. Primitive action plots which interact with (e.g., configure) skill components and are thus specific for skills as they use concrete configuration parameters. Abstract action plots which use primitive action plots but do not directly interact with skills and are thus more general

Abstract action plots are plots that use only primitive action plots. They do not contain communication calls to the skill layer and are independent from it, which makes them more generic. The benefit is that they are highly reusable in other applications or robot platforms with completely different skill components that offer other services but have the same abilities. Since all the state machines offer an interface, only the underlying primitive action plots need to be exchanged or bound before deploying the application.

5.10 Building applications: workflow

The workflow for building applications is illustrated in Fig. 13: Developers provide algorithms in components. These components are bundled with (primitive) action plots that encode basic abilities. E.g. components for map building, collision avoidance and path planning can be bundled and combined with action plots realizing goto tasks and form a bundle navigation. Given that these elements are reusable, they can be provided in a market, illustrated as the “robot app store” in Fig. 13 towards a market.

Task experts can model tasks as abstract action plots (e.g., clean up the table) with parts for later refinement (e.g., with object recognition and driving/goto table). These action plots and behavior packages may again go into the market/“robot app store” or further packaged to full-fledged tasks and again published.

An application expert or system integrator begins with modeling the application specific task plot as abstract action plot (Fig. 13). It is refined in a building-blocks manner by reusing packages, action plots or even full-fledged tasks as off-the-shelf bundles from the market or “robot app store”. If using only provided action plots, the user can put together abstract action plots without having to know about any technical details (configuration, event activations, parametriza-

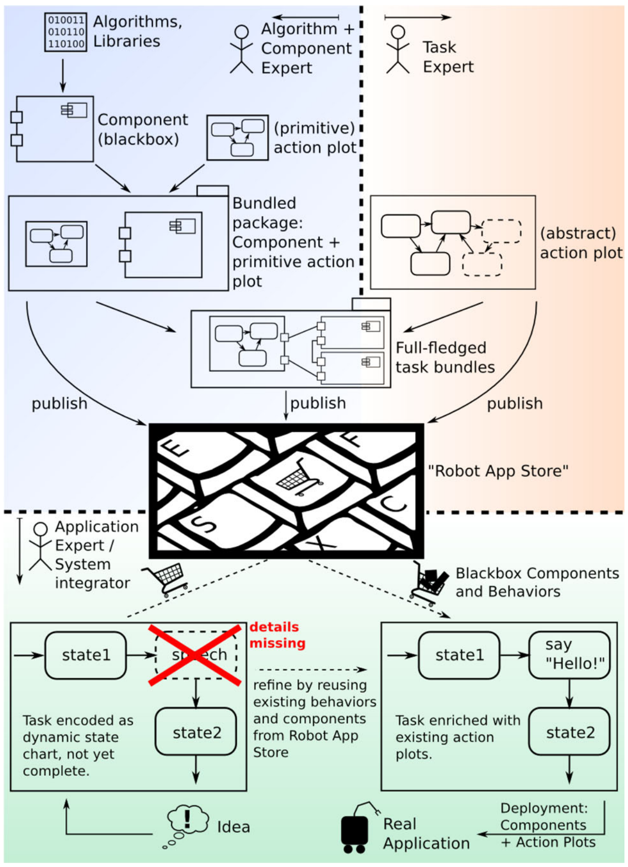

010011
010110
110100
Algorithms, Libraries
Algorithm + Component Expert
Task Expert
Component (blackbox)
(primitive) action plot
Bundled package: Component + primitive action plot
(abstract) action plot
Full-fledged task bundles
publish
publish
publish
"Robot App Store"
Application Expert / System integrator
Blackbox Components and Behaviors
state1
speech
details missing
refine by reusing existing behaviors and components from Robot App Store
state2
Task encoded as dynamic state chart, not yet complete.
state1
say "Hello!"
state2
Task enriched with existing action plots.
Idea
Real Application
Deployment: Components + Action Plots

Fig. 13 A "robot app store" as a repository of algorithms in components, their corresponding (primitive) action plots to configure and use them and task descriptions (abstract action plots) that application experts put together for a specific application. The roles are clearly separated

tions, internal knowledge of algorithms or details of action plots). This way, every role can focus on its contributions to the robotics business ecosystem.

6 Experiment and results

A real-world example is used to demonstrate the approach. Even though the example is small in complexity, it includes very typical use cases of larger applications since the foundations of these use cases are based on insights and experience gained by implementing and operating real-world and complex robot applications like the robot butler scenario [22] where a robot acts as a butler and operates a coffee machine, delivers coffee or fetches juice.

The presented example not only shows how it is implemented from a technical or modeling point of view but also how the steps relate to the development process and separated roles. The example illustrates the benefit of the described approach and development method.

6.1 Experiment description

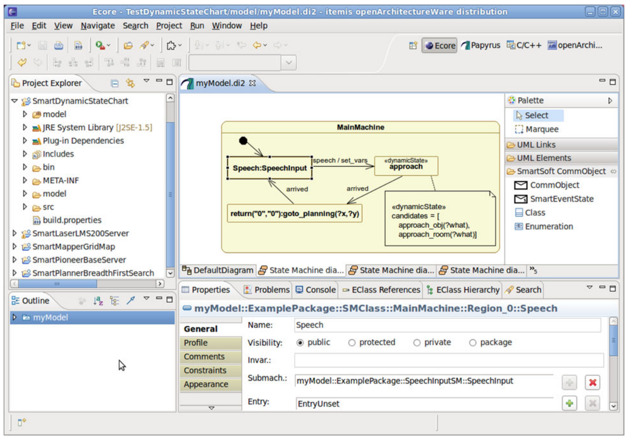
From a story point of view, a robot (Pioneer P3DX platform) is commanded via speech to navigate to rooms (e.g., dining room) or objects (e.g., shelf). It shall return immediately and wait for further commands. Such a task may be used to extend the robot butler scenario [22] as basis for fetch and carry tasks. Objects and rooms have to be approached by a different kind of navigation which is realized through a different configuration and use of different components of the skill layer.

From a development point of view, only the main story (action plot) shall be described as Dynamic State Charts. All (sub) action plots and components required (e.g., for path planning and navigation) shall be reused in the form of packages that bundle components and action plots from a repository.

6.2 Implementation and modeling

The example was created using the SmartSoft MDSD Toolchain [31], in which we integrated a prototypical implementation of Dynamic State Charts (Fig. 14). It uses Papyrus UML [18] for graphical modeling. The state chart is generated to C++ code by the toolchain and can be deployed to the robot.

The deployment diagram in Fig. 15 is used for this example and shows the components and their initial wiring. The Dynamic State Charts acts as sequencer and controls and configures all other components. The SmartPioneerBaseServer wraps the robot hardware, the SmartLaserLMS200Server provides laser scans. In addition, there is a mapper to provide a map, a planner to plan paths and the SmartCdlServer for motion execution and collision avoidance. All components are available at [31].



myModel::ExamplePackage::SMClass::MainMachine::Region_0::Speech

Name: Speech
Visibility: [✓ public] protected private package
Invar.:
Submach.: myModel::ExamplePackage::SpeechInputSM::SpeechInput
Entry: EntryUnset

Fig. 14 A screenshot of the SmartMDSD Toolchain with a prototypical implementation of Dynamic State Charts. The state chart diagram of the example is opened

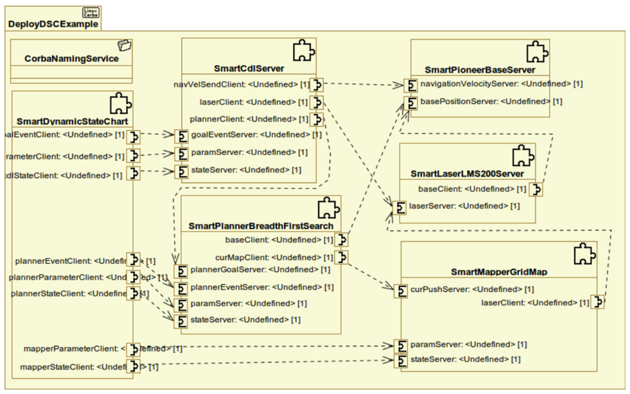


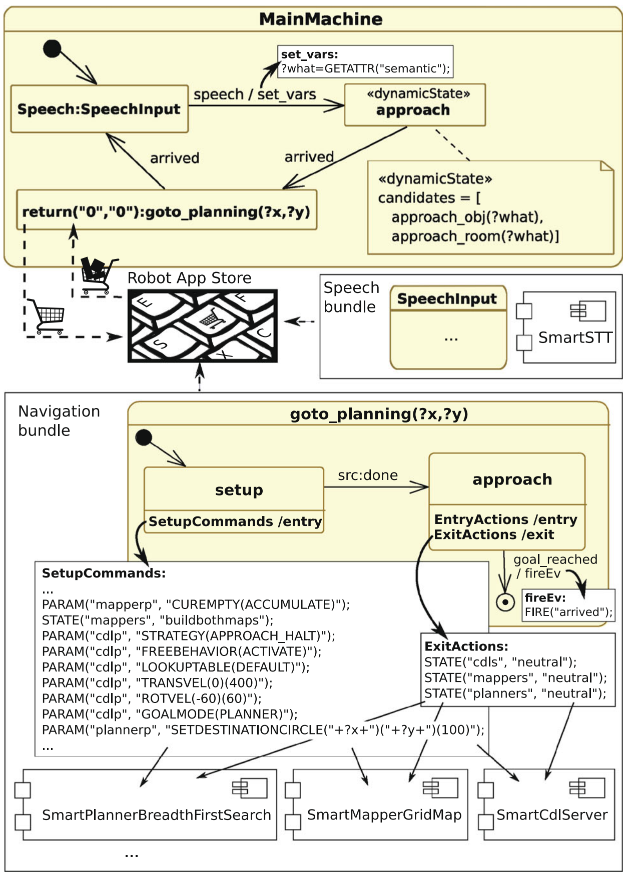
Fig. 15 The deployment diagram shows the components used in the example and their initial wiring

The application is realized as an abstract action plot with three states for speech, approaching and returning to the initial location (Fig. 14). Figure 16 illustrates the modeled Dynamic State Charts MainMachine and illustrates internals of reuse and component parametrization. Only the Dynamic State Charts MainMachine has to be modeled for the example. The states speech and return are submachine states that re-use already existing primitive action plots (e.g., from the “robot app store”). The state approach is a dynamic state with two candidates for approaching rooms or objects that re-uses existing action plots.

The primitive action plots SpeechInput configures components in the skill layer to listen and fire a speech event upon input. As soon as this event is fired, the action set_vars fills the variable ? what with data from the speech event. The object to approach is stored in the speech event and can be accessed by the attribute SEMANTIC.

A dynamic state (approach) is used to decide at run-time how the location given via speech input is approached. Depending on the type (object or room), the dynamic state chooses the instantiation from the candidates approach_obj and approach_room. The candidates know the exact location (e.g., by previously having shown the robot around) and approach the location using a different configuration of the skill layer: an object (e.g., shelf) is expected to be nearby and not to require path planning and is approached using only the SmartCdlServer for this task. Driving to rooms requires more complex path planning and is thus realized by configuring the SmartCdlServer in combination with mapping and path planning. This is done using the primitive action plot *goto_planning* which takes goal coordinates as parameters and handles all details.

The setup state of goto_planning uses the SMARTSOFT state and parameter pattern to configure components in its entry action. It activates the mapper to build maps, configures the collision avoidance (“cdlp”) and finally configures the planner (“plannerp”) with goal coordinates $?x = 0$ and $?y = 0$. The done event is triggered as soon as the substates of



```
set_vars:
?what=GETATTR("semantic");
```

```
«dynamicState»
candidates = [
  approach_obj(?what),
  approach_room(?what)]
```

```
SetupCommands:
...
PARAM("mapperp", "CUREMPTY(ACCUMULATE)");
STATE("mappers", "buildbothmaps");
PARAM("cdlp", "STRATEGY(APPROACH_HALT)");
PARAM("cdlp", "FREEBEHAVIOR(ACTIVATE)");
PARAM("cdlp", "LOOKUPTABLE(DEFAULT)");
PARAM("cdlp", "TRANSVEL(0)(400)");
PARAM("cdlp", "ROTVEL(-60)(60)");
PARAM("cdlp", "GOALMODE(PLANNER)");
PARAM("plannerp", "SETDESTINATIONCIRCLE("+?x+")("+?y+")(100)");
...
```

```
fireEv:
FIRE("arrived");
```

```
ExitActions:
STATE("cdls", "neutral");
STATE("mappers", "neutral");
STATE("planners", "neutral");
```

Fig. 16 Illustrated screenshots showing the main Dynamic State Charts for the overall behavior of the example and a primitive state chart for navigation

a state reach its final state. In this case, there are no substates and done is fired immediately. The entry action of approach activates the event goal_reached in the skill layer. As soon as the event is fired, the state approach is left and it’s exit actions set the components inactive (SMARTSOFT state pattern: “neutral”). approach then fires an internal state machine event arrived which can be used in the upper hierarchy (MainMachine). These configuration commands are specified by the developer (component developer/task expert).

6.3 Results

The example demonstrates a suitable level of abstraction in the modeling of action plots. The overall task is build in a building blocks manner. Existing plots can be reused without knowing about their internal plot or component configuration they use. Therefore, component configuration is decoupled from component usage in action plot modeling. These re-usable plots can be shipped together with components and form a complete behavior package for the idea of the “robot app store” and separation of roles.

However, Dynamic State Charts itself are limited for robot behavior in the way that they can only represent what can be expressed in advance at development time. In use cases where this is not possible or too complex, an external symbolic planner (e.g., Metric-FF [9] or even action recipes from RoboEarth [35]) must be integrated as shown in [34]. They can be used to generate an action plot at runtime and the Dynamic State Charts would control the execution (e.g., the best order to clean up a table given the objects on the table).

Today, powerful tools for model-driven software development exist, especially driven through the Eclipse community (e.g., Xtext, Papyrus UML, YAKINDU). However, these tools are not yet matured enough for all use cases in complex scenarios, since they are still under heavy development and therefore not stable for custom adaptation. For example, YAKINDU supports modeling and code generation for state charts, but both modeling and code generation are not complete. Remarkable progress has been made towards methods to extend and adapt these tools for specific domains, but it still needs more time.

7 Conclusion and future work

We have presented the approach of Dynamic State Charts as a generic method for composition and coordination of complex robot behavior as action plots. The instantiation of dynamic states and the selection of their internals at runtime reduces the complexity in modeling real-world applications and brings robustness by context and situation dependent selection of actions. Modeling action plots is intuitive, since the underlying concept of state charts is known and formally defined.

We have shown how task coordination can be modeled independent from component implementation. These concepts are a progress towards separation of roles in a robotics business ecosystem to make the step from technically driven service robot systems towards use-case driven ones.

Future work includes selection mechanisms for alternative plots and more real-world examples for further evaluation.

Acknowledgments This work has been partially supported by the ITEA2 FIONA project by funding of the German Federal Ministry of Education and Research (BMBF), Förderkennzeichen 01IS13017C. Work on the predecessor was in parts supported by the ZAFH Servicerobotik (http://www.zafh-servicerobotik.de/), with research grants of the state of Baden-Württemberg and the European Union. The authors thank Matthias Lutz and Alex Lotz (project iserveU by funding of BMBF, Förderkennzeichen 01IM12008B) for their support in discussion and reviewing the article.

References

1. Bohren J, Rusu RB, Jones EG, Marder-Eppstein E, Pantofaru C, Wise M, Mosenlechner L, Meeussen W, Holzer S (2011) Towards autonomous robotic butlers: lessons learned with the PR2. In: Proceedings of the IEEE International Conference on Robotics and Automation. Shanghai, China
2. Bruyninckx H (2001) Open robot control software: the OROCOS project. In: Proceedings of IEEE International Conference on Robotics and Automation, vol. 3, pp 2523–2528
3. Bruyninckx H, Klotzbücher M, Hochgeschwender N, Kraetzschmar G, Gherardi L, Brugali D (2013) 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, SAC '13, pp. 1758–1764. ACM, New York, NY, USA. doi:10.1145/2480362.2480693
4. Eclipse Foundation: Xtext Website. http://eclipse.org/xtext. Visited 7 Aug 2013
5. Firby RJ (1989) Adaptive execution in complex dynamic worlds. Ph.D. thesis, Yale University, New Haven, USA
6. Firby RJ (1994) Task networks for controlling continuous processes. In: Proceedings of the Second International Conference on AI Planning Systems
7. Gindele T, Jagszent D, Pitzer B, Dillmann R (2008) Design of the planner of team AnnieWAYs autonomous vehicle used in the DARPA Urban Challenge 2007. In: IEEE Intelligent Vehicles Symposium, pp 1131–1136. IEEE. doi:10.1109/IVS.2008.4621268
8. Harel D (1987) Statecharts: a visual formalism for complex systems. Sci Computer Progr 8(3):231–274
9. Hoffmann J, Nebel B (2001) The FF planning system: fast plan generation through heuristic search. J Artif Intell Res 14:253–302
10. Hurdus JG, Hong DW (2008) Behavioral programming with hierarchy and parallelism in the DARPA urban challenge and robocup. IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems, pp 503–509. doi:10.1109/MFI.2008.4648045
11. Itemis AG: Yakindu Statechart Tools (2013). http://www.yakindu.org. Visited 7 Aug 2013
12. Klotzbuecher M, Bruyninckx H (2012) Coordinating robotic tasks and systems with rFSM statecharts. J Softw Eng Robot (JOSER) 3(1):28–56
13. Larman C (2005) Applying UML and patterns—an introduction to object-oriented analysis and design and iterative development. Prentice Hall, New Jersey
14. Mülder A, Nyßen A (2011) TMF meets GMF. Eclipse Magazin 3:74–78
15. Niemüller T, Ferrein A, Lakemeyer G (2009) A Lua-based behavior engine for controlling the humanoid robot Nao. In: RoboCup Symposium
16. Noreils F (1990) Integrating error recovery in a mobile robot control system. In: Proceedings of the IEEE International Conference on Robotics and Automation, pp 396–401. doi:10.1109/ROBOT.1990.126008
17. Object Management Group (2010) Unified modeling language. Superstructure 2.3
18. Papyrus (2013) http://www.eclipse.org/papyrus/. Visited 7 Aug 2013
19. Pontifcia Universidade Catlica do Rio de Janeiro, Computer Science Department: The Lua Programming Language (2013). http://www.lua.org/. Visited 7 Aug 2013
20. Quigley M, Conley K, Gerkey BP, Faust J, Foote T, Leibs J, Wheeler R, Ng AY (2009) ROS: an open-source robot operating system. In: ICRA Workshop on Open Source Software
21. Risler M (2009) Behavior control for single and multiple autonomous agents based on hierarchical finite state machines. Dissertation, Universität Darmstadt
22. Servicerobotik-Ulm/RoboticsAtHsUlm (2012) The Robot Butler Scenario. YouTube-Video. http://youtu.be/nUM3BUCUnpY. Visited 7 Aug 2013

23. ROS Community: ROS.org wiki: SMACH (2011). http://www.ros.org/wiki/smach/. Visited 7 Aug 2013
24. Schlegel C (2004) Navigation and execution for mobile robots in dynamic environments: an integrated approach. Ph.D. thesis, University of Ulm
25. Schlegel C, Lotz A, Steck A (2011) SmartSoft: the state management of a component. Tech. rep., University of Applied Sciences Ulm
26. Schlegel C, Lutz M, Lotz A, Stampfer D, Inglés-Romero JF, Vicente-Chicote C (2013) Model-driven software systems engineering in robotics: covering the complete life-cycle of a robot. In: Informatik 2013, Workshop Roboter-Kontrollarchitekturen, Springer LNI der GI. Koblenz, Germany
27. Schlegel C, Steck A, Lotz A (2012) Model-driven software development in robotics: communication patterns as key for a robotics component model. In: Chugo D, Yokota S (eds) Introduction to modern robotics. iConcept Press, pp 119–150
28. Schlegel C, Steck A, Lotz A (2012) Robotic software systems: From code-driven to model-driven software development. In: Dutta A (ed) Robotic systems—applications, control and programming. InTech, pp 473–502
29. Simons, AJH (2000) On the Compositional Properties of UML Statechart Diagrams. In: Proceedings of the 2000 International Conference on Rigorous Object-Oriented Methods. Swinton, UK
30. Simons AJH, Graham I (1999) 30 Things that go wrong in object modelling with UML, chap. 17. Kluwer Academic Publishers, pp 237–257
31. SmartSoft at Servicerobotik Ulm (2013). http://www.servicerobotik-ulm.de/. Visited 18 Dec 2013
32. Stampfer D (2010) Dynamic State Charts for task sequencing for service robots. Master's thesis, University of Applied Sciences Ulm, Germany
33. Stampfer D, Schlegel C (2013) Dynamic State Charts: composition and coordination of complex robot behavior and reuse of action plots. In: Proceedings of IEEE Int. Conf. on Technologies for Practical Robot Applications (TePRA). Woburn, MA, USA
34. Steck A, Schlegel C (2011) Managing execution variants in task coordination by exploiting design-time models at run-time. In: Proceedings of IEEE/RSJ Int. Conf. on Robotics and Intelligent Systems (IROS). San Francisco, USA
35. Waibel M, Beetz M, Civera J, D'Andrea R, Elfring J, Galvez-Lopez D, Haussermann K, Janssen R, Montiel JMM, Perzylo A, Schiessle B, Tenorth M, Zweigle O, van de Molengraft R (2011) RoboEarth—a World Wide Web for robots. IEEE Robotics and Automation Magazine, pp 69–82