vTSL – A Formally Verifiable DSL for Specifying Robot Tasks

Christian Heinzemann, Ralph Lange
Robert Bosch GmbH, Corporate Sector Research and Advance Engineering, 71272 Renningen, Germany
firstname.lastname@de.bosch.com

Abstract— **Preprogramming of tasks still plays an important role in complex robotic systems despite the advances in automated planning and symbolic learning. Often, it is desired that end-users implement further tasks to adapt the robotic application to their needs. These user-defined tasks have to meet safety and integrity constraints for protecting the robotic platform and its users. We introduce a verifiable task specification language (vTSL) that enables to automatically prove that a task specification satisfies a set of predefined or task-specific constraints. We illustrate our approach using an example of a self-driving vehicle for intra-logistics and report experiences with two commercial applications.**

I. Introduction

Modern robotic software architectures follow a layered approach. The layer with the core algorithms for SLAM, vision-based object recognition, motion planning, etc. is often referred to as skill layer or functional layer. To accomplish a complex task, these skills are orchestrated by one or more upper layers. Typical names for these layers are planning and executive layer, mission and task layer, or deliberation layer.

The mechanisms being used on these layers are highly application-dependent. They range from preprogrammed tasks (e.g., in the form of finite state-machines, behavior trees or scripts) to automated reasoning, classical planning and symbolic learning.

Advanced robotic applications typically use a mix of these mechanisms and the plenty of robotic FSM editors (e.g., FlexBe [1] or Robot Task Commander [2]), behavior tree frameworks (e.g., ROS behavior_tree [3] and ROS decision_making [4]) and robot task DSLs (e.g., b-script [5] or TDL [6]) illustrates that preprogramming of tasks still plays an important role. In many use-cases it is even desirable that end-users can program (sub-)tasks according to their needs.

Example. As a running example, we consider self-driving vehicles (SDV) for shop-floor areas. An SDV autonomously loads, transports and unloads containers or palettes between the machines, storages and loading ramps. Planning and scheduling of the transport tasks (including the coarse path to drive) is performed by a central fleet management system for all SDVs on the shop-floor. However, the subtasks for loading and unloading are preprogrammed and adapted to installations such as guide rails, markers, and slideways. Other examples for tailored, preprogrammed subtasks are interactions with automatic doors or freight elevators.

Common goals of the aforementioned visual and textual task languages are ease of use, flexibility, clarity and simplicity. This implies that the task interface should hide the complexity of the underlying skill layer as far as possible. In particular for heavy and/or high-priced robotic hardware, it should also prevent any actions that violate safety constraints or that endanger the hardware integrity.

Of course, such constraints can and should be also checked during runtime. Yet, to avoid expensive downtimes, user task specifications should be verified already during the programming. Therefore, we propose the *verifiable Task Specification Language* (vTSL), which is designed for formal verification by model checking with Spin [7]. vTSL is a domain-specific language (DSL) and allows to verify a user task specification against a set of safety and integrity constraints provided with the robot platform as well as against task-specific constraints.

The largest potential for misuse or faulty operation of robotic systems is not in individual parameters or inputs, but in the interplay of the different skills, subsystems and even with the environment. For example, two relevant constraints from SDV applications could be: (1.) The lift of the vehicle must not be actuated while moving. (2.) The vehicle has to stop before turning, except for dedicated free-navigation areas in the map. These two simple constraints already give an idea of the complexity of such formal verification. Therefore, vTSL also provides mechanisms to specify the abstract behavior of the skills as well as the interplay with the environment. In detail, our contributions are as follows:

1) vTSL – a verifiable DSL for specifying task trees for robotic applications
2) A transformation of vTSL models into Promela models to be verified with the Spin model checker
3) Experiments that demonstrate the viability of vTSL and that give an indication on the scalability of the approach.

The remainder of this paper is structured as follows. Sect. II discusses related works. Thereafter, we present the concepts of vTSL in Sect. III and the results of our experiments in Sect. IV, before concluding the paper in Sect. VI.

II. Related work

Nordmann et al. survey domain specific languages (DSLs) for robotics in [8]. In their analysis of 137 DSLs, they identified 59 DSLs that have control and handling of events on the architecture level in focus, which clearly outweighs all other domains such as motion control, kinematics or force control. This supports our observation that preprogramming of task plays an important role – even for modern applications. The survey does not consider verifiability as a distinct property.

Published in Proc. of 2018 IEEE/RSJ Int'l Conference on Intelligent Robots and Systems (IROS), pp. 8308-8314. Madrid, Spain. October 2018.
http://dx.doi.org/10.1109/IROS.2018.8593559

Consequently, we found only few approaches that explicitly consider verifiable DSLs for robotics.

The only verification approach for task trees has been proposed by Simmons et al. [9]. It supports the verification of task activation and synchronization, but does not consider the actual behavior of the tasks.

Armbrust et al. [10] use behavior networks for describing the robots behavior and support their verification using model checking. In contrast to our approach, behavior networks define a set of conditions that define which behavior is activated instead of using explicit control flow. During verification, only activations of behaviors are checked but not their internal behavior.

Cowey and Taylor [11] propose a script language for defining assembly tasks for robot arms. A script can be translated into a specification for the theorem prover Coq. Verification in Coq [12] can only be performed semi-automatically and requires significant expertise in theorem proving.

MissionLab [13] defines a single automaton for a mobile robot where each state of the automaton defines either a sensing operation or a primitive action (e.g., moving). Then, automata of single robots are combined to verify a mission for several robots. The verification is performed using a process-algebra. Our language vTSL focuses on a single robot but supports a more elaborate behavior specification.

Furthermore, there exist several approaches from the domain of cyber-physical systems. These approaches are mainly based on (hierarchical) state machines. Examples include Hugo/RT [14], AADL [15], RT-DEVS [16], and MechatronicUML[17], which even consider quantitative timing properties. However, none of these approaches supports task trees, which we consider to be well suited for robotic applications. In addition, we put a strong focus on readability of vTSL models and allow for using robotic middlewares with different communication semantics.

III. Language concepts

In this section, we first discuss requirements and key ideas for vTSL before we describe the basic modeling concepts and the interfacing with the Robot Operating System (ROS). Finally, we describe our implementation of vTSL.

A. Requirements and key ideas

Verifiability. The most important requirement of our language is to be verifiable. To this end, verifiable means that we can provide a fully automated transformation into a formal verification tool that verifies whether a model being specified in our behavior modeling language satisfies a set of formal requirements. If a feature cannot be translated directly and if the behavior associated with it cannot be abstracted automatically in a meaningful way, we will not include this feature into our language.

Task tree semantics. The language should be easy to use for developers that have experience with the principle of hierarchical task decomposition by task trees such as in the Task Description Language [6] and Hierarchical Task Networks (e.g., [18]).

Expressiveness. The language should be expressive enough to enable the specification of real world robot behaviors without too much abstraction. Therefore, we strive at including as many features of modern programming languages as possible without violating the requirement of being verifiable. In detail, we strive at a textual language that shares some resemblance with C/C++.

Synchronous programming. Synchronous programming languages provide deterministic concurrency for reactive systems. A new representative of these languages is Céu [19]. This textual programming languages features lightweight constructs named par/or and par/and to spawn and join concurrent activities – named *trails* – with deterministic execution semantics. It also includes a mechanism to abort activities from outside.

Our language shall provide similar mechanisms to call subtasks concurrently and to abort sibling task trees. We do not aim at the strong determinism semantics of synchronous programming languages here, but require at a lightweight mechanism to prevent races between concurrent subtasks on shared data. One such mechanism is cooperative multitasking with coroutines.

Interfacing with skill-layer components. The language should enable to interface with skill-layer components of the robot system that are orchestrated by the behavior model, i.e., it should be possible to send data to components and to retrieve data from components or to wait for events from components. Since skill-layer components can typically not be included into a verification procedure based on their C++ implementation, the new behavior modeling language should enable to provide abstract replacement models for components, possibly via providing stubs. As a primary robotic middleware, we strive at supporting ROS.

Source code generation. Task trees specified by the behavior modeling language should be directly generatable into an implementation that can be executed on the robot.

B. Task tree modeling

This section informally describes the elements of vTSL, whose semantics is defined in form of a translational semantics by the mapping to Promela described in Section IV.

The basic building block of vTSL is an *action*. An action is an executable behavior that may invoke other actions and that may be executed concurrently to other actions.

For the remainder of the paper, we use the action `FindLoadingBay` shown in Figure 2 as an example to

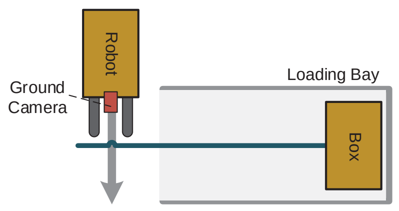
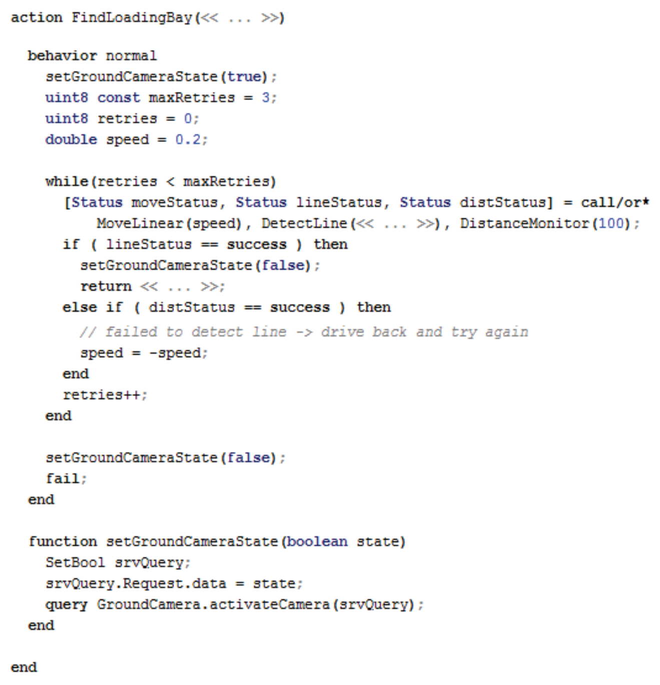


Fig. 1. Illustration of FindLoadingBay



```
action FindLoadingBay(<< ... >>)

  behavior normal
    setGroundCameraState(true);
    uint8 const maxRetries = 3;
    uint8 retries = 0;
    double speed = 0.2;

    while(retries < maxRetries)
      [Status moveStatus, Status lineStatus, Status distStatus] = call/or*
          MoveLinear(speed), DetectLine(<< ... >>), DistanceMonitor(100);
      if ( lineStatus == success ) then
        setGroundCameraState(false);
        return << ... >>;
      else if ( distStatus == success ) then
        // failed to detect line -> drive back and try again
        speed = -speed;
      end
      retries++;
    end

    setGroundCameraState(false);
    fail;
  end

  function setGroundCameraState(boolean state)
    SetBool srvQuery;
    srvQuery.Request.data = state;
    query GroundCamera.activateCamera(srvQuery);
  end

end
```

Fig. 2. Example of an advanced action

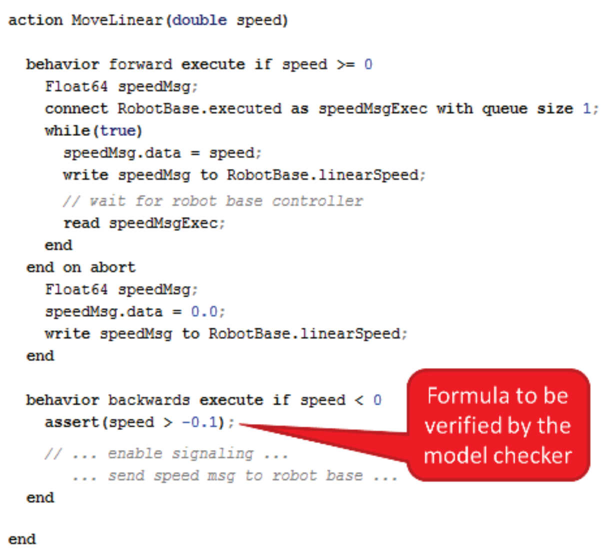
illustrate the main concepts of vTSL. The implemented behavior is illustrated in Fig. 1. The SDV has the task to load the box that is located in the loading bay. With `FindLoadingBay`, the SDV searches for a blue marker line of the loading bay using its ground camera. More precisely, the SDV moves orthogonal to the line (via action `MoveLinear`, cf. Fig. 3) until it detects the line (via parallel action `DetectLine`, cf. Fig. 4) or until it has traveled too far (via parallel action `DistanceMonitor`). In the latter case, the SDV will drive backwards and retry to detect the blue line.

An action in vTSL is identified by the keyword `action` followed by the name of the action and two parameter lists. These are the input parameters and the return types. The input parameters are passed to the action when it is called and cannot be modified during execution of the action. The optional list of return types is listed after a colon. If the list of return types is omitted, the action has return type void.

The action body has a fixed structure: First comes a list of variable declarations, which can be used in the entire action.

Second comes one or more `behavior` blocks that implement the behavior of the action. Each action has at least one behavior. If it defines more than one behavior, each behavior has to define a condition under which it is executable. The condition may only refer to the input parameters of the action. The behavior whose condition is satisfied will be executed. In our example, the action `MoveLinear` in Fig. 3 has two behaviors: one for driving forward and one for driving backwards, where the execute condition checks whether the desired speed is positive or negative.

Third comes a list of functions that enables to implement private helper functions that may be called from behaviors



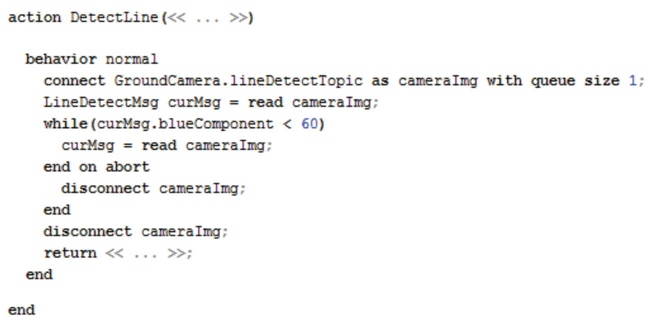
```
action MoveLinear(double speed)

  behavior forward execute if speed >= 0
    Float64 speedMsg;
    connect RobotBase.executed as speedMsgExec with queue size 1;
    while(true)
      speedMsg.data = speed;
      write speedMsg to RobotBase.linearSpeed;
      // wait for robot base controller
      read speedMsgExec;
    end
  end on abort
    Float64 speedMsg;
    speedMsg.data = 0.0;
    write speedMsg to RobotBase.linearSpeed;
  end

  behavior backwards execute if speed < 0
    assert(speed > -0.1);
    // ... enable signaling ...
       ... send speed msg to robot base ...
  end

end
```

Fig. 3. Example of an action with multiple behaviors.

and other functions within the same action. As an example, consider the function `setGroundCameraState` in Fig. 2.

The behaviors and functions contain a block with imperative statements that follow the standard semantics of C/C++. In particular, we support the regular if-statements, for-loops, while-loops, and variable declarations. We support the basic C expressions (arithmetic operators, logical operators, comparison operators, bitwise operators, shifts) and all primitive types of C (void, boolean, char, short, int, long, float, double) in their signed and unsigned variants including the C99 types with explicit bit length (e.g., int8, int16).

A core feature of the DSL are subaction calls that enable to invoke further actions from the behavior of an action. Subaction calls may invoke an arbitrary number of subactions at once. If more than one subaction is invoked, the call semantics must be defined. The call-semantics is based on Céu's par/and and par/or semantics. For and-semantics by `call/and*`, the call returns after all subactions have returned. For or-semantics by `call/or*`, the call returns after one subaction has returned. In the latter case, all other subactions belonging to the call are aborted, but the invoking action waits until all aborts have been executed completely. Furthermore, an action provides a return status that indicates



```
action DetectLine(<< ... >>)

  behavior normal
    connect GroundCamera.lineDetectTopic as cameraImg with queue size 1;
    LineDetectMsg curMsg = read cameraImg;
    while(curMsg.blueComponent < 60)
      curMsg = read cameraImg;
    end on abort
      disconnect cameraImg;
    end
    disconnect cameraImg;
    return << ... >>;
  end

end
```

Fig. 4. Example of a monitoring action waiting for an event.

whether its execution was successful, whether it failed, or whether it was aborted. The return status can be used in the calling action, for example to initiate retries as done in `FindLoadingBay` in Fig. 2.

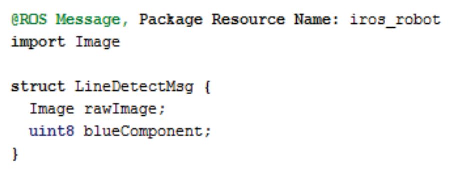
On a programming level, aborting an action is comparable to an exception happening. To enable graceful shutdown of an aborted action, we support so-called abort handlers, which are entered if the action is aborted. They allow to bring the underlying skill components into a consistent state (e.g., to stop a hardware interaction).

We also support `par/and` and `par/or` statements within behaviors as lightweight alternatives to subaction calls. Initially, the branches are triggered from top to bottom. `par/and` terminates after all branches have terminated, while `par/or` terminates after one branch has terminated.

Typically, `call/and*` and `par/and` are used for starting operations on skill layer components in parallel, which all shall be finished. `call/or*` and `par/or` are often used for starting an operation and one or more monitors, where the execution of the operation is aborted if a monitoring condition is satisfied.

C. Interfacing with Robotic Middleware

vTSL contains special keywords for interfacing with skill-layer components running on a robotic middleware. In the scope of this paper, we focus on ROS and show how to define ROS messages, ROS topics, and ROS services. ROS messages are basically represented as C structs. The DSL ships with all message types from std_msgs and common_msgs (e.g., geometry_msgs), but it also allows to define custom message types like `LineDetectMsg` in Fig. 5.



```
@ROS Message, Package Resource Name: iros_robot
import Image

struct LineDetectMsg {
  Image rawImage;
  uint8 blueComponent;
}
```

Fig. 5. Example of custom ROS message type

ROS topics are supported by dedicated keywords for connecting to topics (`connect`), for writing messages to topics (`write`), for reading messages from topics (`read`), and for disconnecting from topics (`disconnect`). A typical sequence of statements for connecting to a topic and reading a message is given in Fig. 4 where a `LineDetectMsg` is read from the `lineDetect` topic of the `GroundCamera`.

ROS services are defined based on service declarations that follow the common ROS structure with request and response message. vTSL provides a special keyword `query` for invoking a ROS service. As usual, the values of the return message are available in the response field of the service object and `query` returns whether the service execution was successful or not. An example is given in Fig. 2 where the ground camera is activated by a ROS service query.

The semantics of the constructs is as follows: Reading from a topic always causes the action that performs the read to yield, i.e., another action that is ready for execution will be executed. If a message is present on the topic being read, the action remains ready. If no message is present on the topic, then the action is blocked and will only become ready again after a message has been written on the topic. Invoking a service is implemented with the semantics of a remote procedure call, i.e., it blocks the entire task tree until it returns as all actions of the task tree are executed in a single task according to the synchronous semantics.

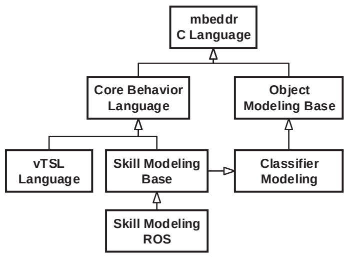


Fig. 6. Implemented Languages in MPS.

D. Implementation with MPS

We have implemented our vTSL in the language workbench MPS[1] using the mbeddr platform.[2] MPS is based on projective editing, i.e., the editor directly displays a projection of the underlying model making it unnecessary to implement a parser. A particular focus of MPS is composability of languages. We split the vTSL implementation into several sub-languages as shown in Fig. 6.

The *mbeddr C language* ships with the mbeddr platform. From this language, we use all basic C types and expressions. We extend the C type system by objects (*Object Modeling Base*) and a classifier concept (*Classifier Modeling*), which shall support object-oriented features in the future, but only defines struct types at present. The *Core Behavior Language* defines the basic statements of vTSL and additional types like strings and arrays. Based on this, we define separate language for defining actions and components. For components, we provide a base implementation in *Skill Modeling Base* and a ROS specific extension in *Skill Modeling ROS*. This enables to provide an integration of vTSL for a different robotic middleware easily as an extension.

IV. Verification by Model Checking

In order to establish verifiability of our DSL, we provide a model transformation that translates a model written in vTSL into a Promela model for the open source model checker Spin [7]. This model includes the actions of the task tree and a set of component stubs described next that represent the behavior of the underlying robots skills for the verification.

A. Components stubs

A major goal of vTSL is to verify the interaction of the actions in the task tree with the underlying components on

[1]`https://www.jetbrains.com/mps/`
[2]`http://mbeddr.com/`

```
skill component GroundCamera {

  Advertised Topics:
    topic LineDetectMsg lineDetectTopic;

  Offered Services:
    service SetBool activateCamera;

  initialization
    boolean cameraState = false;
    SetBool.Request activateCamQuery;
    SetBool.Response activateCamResp;
  end

  execute behavior
    choose
      [activateCamQuery, activateCamResp] = receive GroundCamera.activateCamera();
      cameraState = activateCamQuery.data;
      activateCamResp.success = true;
      reply GroundCamera.activateCamera(activateCamResp, true);
    or
      (cameraState == true);
      LineDetectMsg lineMsg;
      choose
        lineMsg.blueComponent = 59;
      or
        lineMsg.blueComponent = 61;
      end
      write lineMsg to GroundCamera.lineDetectTopic;
    end
  end
}
```

Fig. 7. Structure of skill component GroundCamera

the skill layer. Therefore, we model stubs for all components including the communication with ROS topics and services. For each component stub, we implement the I/O behavior of the component and an abstraction of its internal state.

A component stub always has the following fixed structure as illustrated in Fig. 7 for `GroundCamera`: First comes a list of advertised topics. These are topics that the component reads or writes. Second comes a list of offered services. These services may be called by actions from the task tree or by other components. Together, advertised topics and offered services define the interface of the component. Third comes an `initialization` section. In this section, all variables that determine the (abstraction of) the inner state of the component are declared and initialized. Fourth and finally comes an `execute behavior` section that determines the I/O behavior of the component.

The execute behavior section typically consists of one infinite while loop with a choose statement inside. A choose statement represents a non-deterministic choice where in each execution, one of the branches is chosen for execution. However, a branch may only be chosen if it is actually executable, whereas a branch is considered to be executable when the first statement is executable. This semantics is in line with the semantics of choices in Promela. For representing the I/O behavior of the component, we use one branch for each read topic and one for each offered service.

Please note that while the task tree is implemented on a level of detail that enables to automatically generate code for the robot, the components are only stubs to be used for verification. In order to verify all possible execution traces, we use non-deterministic choices for possible results of computations in component stubs as, for example, in case of the line detection in Fig. 7. Also, the stubs do not necessarily correspond with individual ROS nodes. A stub may also represent a namespace with a group of nodes.

B. Mapping to Promela

Promela, which is the input language of the Spin model checker [7], is a textual language that uses *proctypes* as main modeling element. Proctypes have parameters and can be instantiated to processes, each having a unique process identifier (pid). The body of a proctype defines its behavior and is implemented in an imperative fashion. Fig. 8 shows a slightly simplified version of the generated Promela model to give an impression on the structure of the generated models.

In our transformation a proctype for each action and for each component stub in vTSL is created. The body of the proctype contains the different behaviors of the action that are combined using an if-statement. The transformation of expressions and statements is straightforward as Promela has some resemblance with C. Assertions in the DSL are translated to assertions in Promela. Functions are implemented by using helper macros of Promela.

The synchronous semantics of action execution is controlled by a set of helper variables exemplified at the top of Fig. 8. First, `runningAction` contains the pid of the currently scheduled action. Each action, which is currently blocked, blocks on a condition that waits until its own pid equals the `runningAction`. The currently reading and blocked actions are maintained in two arrays that are controlled by a so-called dispatcher proctype. The dispatcher becomes active each time an action blocks. In this case, the `runningAction` contains the pid of the dispatcher. Then, the dispatcher selects the next action out of the `readyQueue` and makes it the running action. The action

```
byte runningAction;
byte readyQueue[DISPATCHER_QUEUE_SIZE];
byte blockedQueue[DISPATCHER_QUEUE_SIZE];
byte abortSubaction[2];

proctype FindLoadingBay(){
  byte childPid, childPid2;
  { //wait for run signal
    (runningAction == _pid);

    //user defined action code starts here!
    _FindLoadingBay_setGroundCameraState(true);
    #define maxRetries 3
    byte retries = 0;

    do
    :: retries < maxRetries -> {
       mtype distStatus;
       childPid = run gen__lambda__IROS_Gen_FindLoadingBay_1();
       childPid2 = run IROS_Gen_DistanceMonitor(100);
       callParallelSubactions(childPid, childPid2);
       awaitParallelORSubactionReturn(childPid, childPid2, _, distStatus);
       (runningAction == _pid);
       if
       :: IROS_Gen_FindLoadingBay_normal_lineStatus == SUCCESS -> {
          _IROS_Gen_FindLoadingBay_setGroundCameraState(false);
          ActionReturn(SUCCESS); }
       :: else -> { ... }
       fi;
       retries++; }
    :: else -> { break; }
    od;

    _FindLoadingBay_setGroundCameraState(false);
    ActionReturn(FAILED);
    //user defined action code ends here!
  } unless {
    {(abortSubaction[0] == _pid || abortSubaction[1] == _pid);
    (runningAction == _pid);
      ActionReturn(ABORTED);
    };
  };
  ActionReturn(SUCCESS);
}
```

Fig. 8. Generated Promela Model for `FindLoadingBay`

that yielded is transferred into the `blockedQueue`.

The sample action `FindLoadingBay` invokes three subactions. In our model transformation, we normalize such subaction calls such that at most two subactions are contained in a single call by introducing additional lambda actions. Calls are implemented by a set of helper macro definitions that we omit here. Basically, they insert the called subactions into the `readyQueue` and move the calling action into the `blockedQueue`. Since the subactions are called based on par/or semantics, we have to abort all sibling actions once one of the called subactions returns. In our Promela model, we abort a subaction by writing its pid into the `abortSubaction` array. This triggers the condition of the unless clause at the bottom of each action, which in turn causes the main body to be left immediately. This resembles the intended semantics of an abort exception. In addition, the unless clause contains the abort handler.

ROS messages are implemented by Promela typedefs that are very similar to C struct types. Topics are implemented based on so-called channels. A channel resembles a message queue to which processes can write information and from which they can read information. Thereby, messages send to a channel may contain fields of different data types making them a good fit for representing topics. As for ROS topics, processes will block if they attempt to read from an empty topic. ROS services are realized by a pair of two channels; one for the query message and one for the response message.

The transformation from vTSL to Promela has been realized using model-to-model transformation in MPS based on our implementation described in Sect. III-D.

C. *Verification with Spin*

We verify the exported Promela model by loading it into the Spin model checker. By default, Spin will verify that the Promela model does not contain deadlocks and that all assertions are satisfied on all possible execution traces. In addition, Spin supports to define temporal properties for expressing more complex constraints – either specified as linear temporal logic (LTL) formula or as so-called never claim. On the level of vTSL, we currently only support using assertions, which may be placed anywhere a statement is allowed. More complex temporal properties based on LTL need to be specified directly in the Promela model. Such LTL properties, however, could also be shipped along with the robotic platform software.

In our example, the verification yields an error that results from a violated assertion in the action `MoveLinear`. In case that the SDV misses the blue marker line and retries, it simply drives backwards with -0.2 m/s. This, however, violates the assertion that speed is limited to -0.1 m/s for driving backwards in `MoveLinear`.

V. Experiments

This section provides information on our experiences of applying vTSL to a logistics use case (Sec. V-A) and discusses results from a benchmark model demonstrating the scalability of the verification (Sec. V-B).

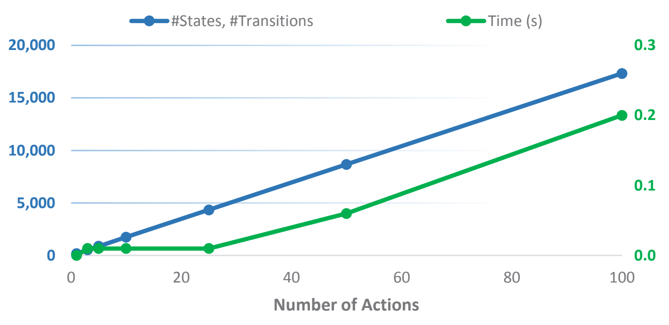


Fig. 9. Increase in number of states and transitions as well as computing time for increasing number of actions.

A. *Experiences with use-case from logistics*

Two years before vTSL, we developed a C++ library that implements vTSL's task tree semantics with call/and and call/or by coroutines. The abort mechanism is implemented by the use of C++ exceptions. The library also provides ROS bindings, but no support for formal verification. This library was used successfully in two commercial logistic robot projects in the Bosch Group.

Our experiences with the corresponding C++-based task trees significantly influenced the concept and design of vTSL. We re-implemented relevant parts of the task trees in vTSL and conducted various experiments with them. For example, using the Spin-based verification, we identified timing problems between activations in the skill components caused by concurrent actions in the task trees.

The developers of the aforementioned logistic robot projects confirmed the improved readability – as to be expected from a DSL. vTSL reduced the code size by more than 50% compared to the C++-based task trees. Our colleagues particularly appreciated the concise syntax for lightweight concurrency by `par/and` and `par/or` and the syntax for abort handling, which are implemented by lambda expressions (i.e. `[&](){...}`) and `try-catch` in C++.

B. *Performance evaluation of verification*

For better estimating the viability of model checking of vTSL specifications, we created a benchmark model in which we can modify the number of actions, the number of variables and statements per action, as well as the number of components. We evaluated this model using Spin v6.4.5 on an HP laptop with Intel Core i5 processor.

First, we analyze the effect of an increasing number of actions on the size of the resulting state space in Spin, which is given by the number of states and transitions. In the experiment, we use 50 statements and 10 variables for each action and increase the number of actions from 1 to 100. Each action invokes two subactions using `call/and*` until the intended number of actions has been launched. The results in Fig. 9 show that the increase is linear in the number of actions (left y-axis, blue) and computing time (right y-axis, green) as we expect due to the synchronous semantics of vTSL. Since the number of states and transitions

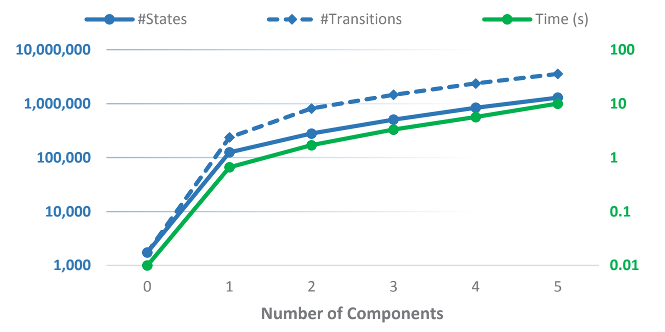


Fig. 10. Increase in number of states and transitions as well as computing time for increasing number of components.

is coincidentally equal in our experiment, we plotted only a single line for states and transitions.

Second, we analyze the effect of an increasing number of components on the size of the state space in Spin. In the experiment, we use ten actions with ten statements and ten variables in each action. Each action invokes two subactions using `call/and*` until the intended number of actions has been launched. The first n leaf actions connect to a topic of a component, wait for a message, and terminate. The results shown in Fig. 10 show that the increase in states and transitions (left y-axis, blue) is exponential in the number of components as we expect due to the fact that components run concurrently to the task tree. The plot also shows that the computing time (right y-axis, green) depends nearly linearly on the number of transitions. The memory consumption is roughly about 300 byte per transition in this experiment, which gives $1.15\,\mathrm{GB}$ for the run with five components.

VI. Conclusions

We presented a DSL, called vTSL, for task trees that allows to verify the specified behaviors against predefined and task-specific constraints of the robot platform using an automated translation to the Spin model-checker. We explained the major concepts of the language, including the mechanisms for concurrency between tasks, for concurrency within tasks and for the abort of subtasks in particular. We showed the scalability in two experiments and reported experiences from commercial use-cases.

Currently, we work on a code generation from vTSL to C++, using the C++ task library mentioned in Sec. V-A. Furthermore, we are working an exception mechanism that allows to escalate exceptions along the task hierarchy. While vTSL currently only supports simple assertions, we need to integrate support for LTL properties and evaluate their impact on the verification time.

Advanced features planned for the next years are support of unbounded container types and verification with timed model checking. While the former can be often circumvented in practice, the latter is essential for continuous processes, which are prevalent in robotics. We also consider to publish vTSL as open-source software – in particular together with potential academic partners conducting research in the last mentioned domains.

References

[1] P. Schillinger, S. Kohlbrecher, and O. von Stryk, “Human-Robot Collaborative High-Level Control with Application to Rescue Robotics,” in *Proc. of the IEEE Int’l Conference on Robotics and Automation (ICRA ’16)*, Stockholm, Sweden, May 2016, pp. 2796–2802.

[2] S. Hart, P. Dinh, J. D. Yamokoski, B. Wightman, and N. Radford, “Robot Task Commander: A Framework and IDE for Robot Application Development,” in *Proc. of the 2014 IEEE/RSJ Int’l Conference on Intelligent Robots and Systems (IROS ’14)*, Chicago, Il, USA, Sept. 2014.

[3] M. Colledanchise and P. Ögren, “How Behavior Trees Modularize Hybrid Control Systems and Generalize Sequential Behavior Compositions, the Subsumption Architecture, and Decision Trees,” *IEEE Transactions on Robotics*, vol. 33, no. 2, pp. 372–389, Apr. 2017.

[4] CogniTeam Ltd., “ROS decision_making,” 2015, retrieved February 27, 2018, from http://wiki.ros.org/decision_making.

[5] T. J. de Haas, T. Laue, and T. Röfer, “A Scripting-based Approach to Robot Behavior Engineering Using Hierarchical Generators,” in *Proc. of the IEEE Int’l Conference on Robotics and Automation (ICRA ’12)*, St. Paul, MN, USA, May 2012, pp. 4736–4741.

[6] R. Simmons and D. Apfelbaum, “A Task Description Language for Robot Control,” in *Proc. of the 1998 IEEE/RSJ Int’l Conference on Intelligent Robots and Systems (IROS ’98)*, vol. 3, Victoria, B.C., Canada, Oct. 1998, pp. 1931–1937.

[7] G. J. Holzmann, *The Spin Model Checker – Primer and Reference Manual*. Addison Wesley, 2004.

[8] A. Nordmann, N. Hochgeschwender, D. Wigand, and S. Wrede, “A Survey on Domain-Specific Modeling and Languages in Robotics,” *Journal of Software Engineering for Robotics (JOSER)*, no. 7, 2016.

[9] R. Simmons, C. Pecheur, and G. Srinivasan, “Towards automatic verification of autonomous systems,” in *Proc. of the IEEE/RSJ Int’l Conference on Intelligent Robots and Systems (IROS 2000)*, vol. 2, Oct. 2000, pp. 1410–1415.

[10] C. Armbrust, L. Kiekbusch, T. Ropertz, and K. Berns, “Tool-assisted Verification of Behaviour Networks,” in *Proc. of the IEEE Int’l Conference on Robotics and Automation (ICRA ’13)*, May 2013, pp. 1813–1820.

[11] A. Cowley and C. J. Taylor, “Towards language-based verification of robot behaviors,” in *Proc. of the IEEE/RSJ Int’l Conference on Intelligent Robots and Systems (IROS ’11)*, Sept. 2011, pp. 4776–4782.

[12] A. Chlipala, *Certified Programming with Dependent Types – A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013.

[13] D. Lyons, R. Arkin, S. Jiang, D. Harrington, and T. Liu, “Verifying and validating multirobot missions,” in *Proc. of the IEEE/RSJ Int’l Conference on Intelligent Robots and Systems (IROS ’14)*, Sept. 2014, pp. 1495–1502.

[14] A. Knapp, S. Merz, and C. Rauh, “Model Checking – Timed UML State Machines and Collaborations,” in *Formal Techniques in Real-Time and Fault-Tolerant Systems*, ser. Lecture Notes in Computer Science, W. Damm and E.-R. Olderog, Eds. Springer Berlin Heidelberg, Sept. 2002, vol. 2469, pp. 395–416.

[15] A. Johnsen, K. Lundqvist, P. Pettersson, and O. Jaradat, “Automated Verification of AADL-Specifications Using UPPAAL,” in *Proc. of the 14th IEEE Int’l Symposium on High-Assurance Systems Engineering*, Oct. 2012, pp. 130–138.

[16] A. Furfaro and L. Nigro, “Embedded Control Systems Design based on RT-DEVS and temporal analysis using UPPAAL,” in *Proc. of the Int’l Multiconference on Computer Science and Information Technology (IMCSIT ’08)*, Oct. 2008, pp. 601–608.

[17] C. Gerking, S. Dziwok, C. Heinzemann, and W. Schäfer, “Domain-specific Model Checking for Cyber-physical Systems,” in *Proc. of the 12th Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVa ’15)*, Ottawa, Sept. 2015, pp. 18–27.

[18] D. Nau, T.-C. Au, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman, “SHOP2: An HTN Planning System,” *Journal of Artificial Intelligence Research*, vol. 20, no. 1, pp. 379–404, Dec. 2003.

[19] F. Sant’Anna, “Structured Synchronous Reactive Programming with Céu,” in *Proc. of the 14th Int’l Conference on Modularity (MODULARITY ’15)*, Fort Collins, CO, USA, Mar. 2015, pp. 29–40.