Template Decision Diagrams for Meta Control and Explainability

Author's Version – 30 April 2024

Clemens Dubslaff[1,2], Verena Klös[2], and Juliane Päßler[3]

[1] Eindhoven University of Technology, Eindhoven, The Netherlands
c.dubslaff@tue.nl
[2] Centre for Tactile Internet with Human-in-the-Loop (CeTI), Dresden, Germany
verena.kloes@tu-dresden.de
[3] University of Oslo, Oslo, Norway
julipas@ifi.uio.no

Abstract. *Decision tree classifiers (DTs)* provide an effective machine-learning model, well-known for its intuitive interpretability. However, they still miss opportunities well-established in software engineering that could further improve their explainability: separation of concerns, encapsulation, and reuse of behaviors. To enable these concepts, we introduce *templates* in *decision diagrams (DDs)* as an extension of multi-valued DDs. Templates allow to encapsulate and reuse common decision-making patterns. By a case study from the autonomous underwater robotics domain we illustrate the benefits of template DDs for modeling and explaining *meta controllers*, i.e., hierarchical control structures with underspecified entities. Further, we implement a template-generating refactoring method for DTs. Our evaluation on standard controller benchmarks shows that template DDs can improve explainability of controller DTs by reducing their sizes by more than one order of magnitude.

Keywords: Decision Diagrams, Decision Trees, Explainability, Control

1 Introduction

Cyber-physical systems are complex. Their behaviors may depend on different kinds of sensor data, have to deal with concurrent executions, and face heterogeneous components that all can be subject to uncertainty such as noise or partial observability. *Controllers* are usually used to steer their behaviors and ensure correct functioning, e.g., in safety-critical environments. Handcrafted engineering of controllers is error-prone and does not scale to large systems. Differently, provably correct controllers can be constructed from formal specifications by *automated controller synthesis* [47]. Due to recent demands by legal authorities [60,1,2,3] and to increase overall trustworthiness in systems, synthesized controllers should be *explainable*.

Decision Trees for Strategy Explanation. *Decision trees (DTs)* [51,46] are

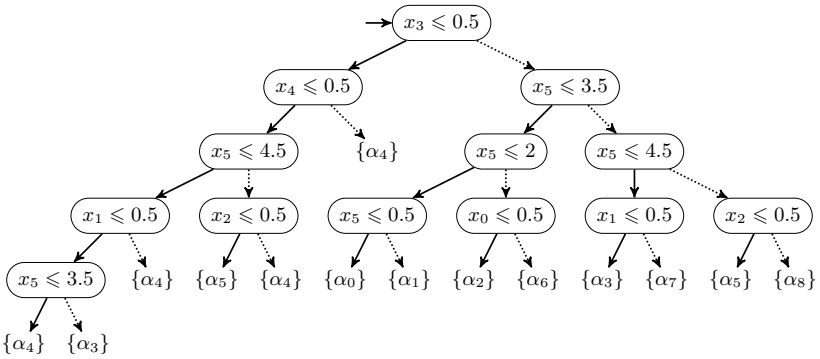

Fig. 1: DT representation of a control strategy for the TRIANGLE benchmark; solid edges are taken if the predicate is fulfilled, dotted edges if it is not.

an effective machine-learning model that also gained attention as explications of synthesized controllers due to their simplistic and interpretable nature [27,5]. Figure 1 shows an example DT of a controller for the probabilistic planning system TIREWORLD [41] that has been synthesized by the tool DTCONTROL [5]. In its basic form, DTs are binary trees where inner *decision nodes* are labeled by predicates over features such as observable system variables and where leafs are labeled with control actions. Each decision node has two outgoing edges for the predicate being fulfilled or not. By starting in the root of the DT and evaluating predicates according to the current state of the controlled system, DTs eventually lead to possible control actions of the controller. Depending on how the controller is synthesized, these actions all fulfill the controller specification, e.g., following its safety requirements.

Due to the intuitive evaluation of predicates in decision nodes, DTs are well-suited to represent and explain the step-wise reasoning in control policies. *Occam's razor* provides a major principle for explainability for a global perspective, according to which the best explanation for a phenomenon is the smallest one amongst all possible explanations [28]. For this, state-of-the-art learning algorithms minimize the size of DTs by choosing suitable decision predicates by entropy-based splitting and placing important decisions first. However, this minimization is only done within the class of DTs, which structurally hinders further improvements for explainability:

1. DTs are trees, duplicating shared decision making as isomorphic subtrees
2. DTs are monolithic, barely separating concerns [21]
3. DTs are deterministically evaluating predicates, only supporting nondeterminism in control actions of DT leafs

In this paper, we introduce the concept of *templates* in *decision diagrams (DDs)*, leading to a new data structure of *template DDs* that remedies the three drawbacks of DTs above. First, template DDs are diagrams instead of trees, allowing for sharing of decision making by multiple incoming edges to

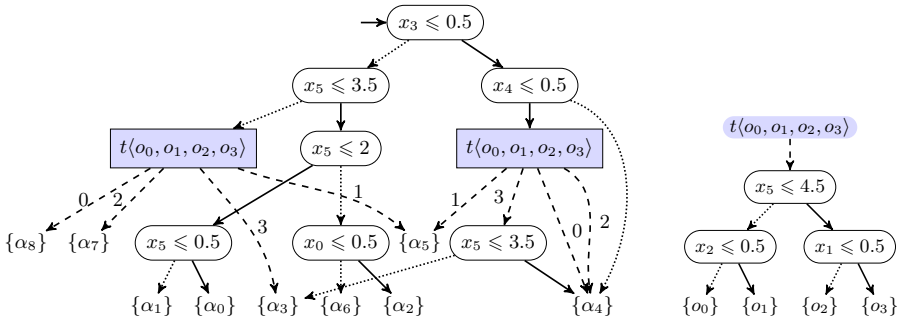

Fig. 3: Template DD for TRIANGLE control strategy

decision nodes similar as within *binary decision diagrams (BDDs)* [12]. Therefore, common decision making but also common control actions can directly be deduced from the graphical representation as diagrams. Second, extending the first reasoning by merging isomorphic subtrees, we propose to even further reuse common decision making by introducing *box nodes*. Such nodes are labeled by a template that itself can stand for decision making modeled as template DDs, hence separating concerns similar to procedures in programming languages by encapsulating common behaviors. The hierarchical structure through boxes and assigned templates does not only improve explainability of control policies by allowing for more compact representation, they also enable *underspecification*, enhancing degrees of freedom. Decision making that is, e.g., irrelevant for ensuring safety but important for other requirements such as energy consumption, could hence be left out for explicating safety under *all possible implementations* of templates. To this end, template DDs are also suitable to specify *families of controllers*, enabling their explanation and family-based analysis [18,17,49]. Figure 3 shows a template DD for the DT controller of Figure 1, reusing common decision making in a template used twice in the diagram.

Meta Control. Many cyber-physical systems are subject to uncertainty in both their environment and their internal structure. The reasons could be, e.g., partial observability, hardware or software failures, or noisy sensor data. To increase reliability, the systems can be implemented as *self-adaptive systems (SASs)* [62], enabling them to adapt their structure, task plan, control strategies, etc., mostly according to a predefined strategy. SASs that adapt their control strategy itself include a controller steering the adaptation in the system and can therefore be considered as a *meta control* system. One can distinguish between *internal* and *external* self-adaptation [62,54].

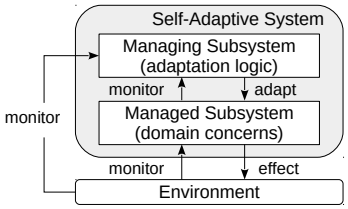

Fig. 2: A two-layered architecture for an SAS [49]

An internal self-adaptation mechanism is embedded in the application logic and realized by, e.g., exception handling or fault-tolerance mechanisms. External self-adaptation separates the application logic from the adaptation logic and introduces a two-layered system where the application logic, focussing on the domain concerns, is implemented in a *managed subsystem* (the control system), and the adaptation logic is implemented in a *managing subsystem* [35] (the meta control system) – see Figure 2. The managed subsystem monitors the environment and may affect the environment with its actions. It can also contain internal self-adaptation like exception handling. The managing subsystem receives input from both the managed subsystem and the environment to determine if and which adaptation is needed, and adapts the managed subsystem if necessary.

We focus here on easing the design and comprehension of meta control systems by distinguishing between high-level views for meta control and low-level means of adaptability. For this, we exploit the template DD's possibility of *underspecification*, realized by (i) leaving the specification of templates open and specifying families of controllers, and by (ii) introducing parameters in box types to steer template instantiations. The templates and parameter values can then be set by the meta controller to adapt the control strategy of the control system.

Contributions. We present three main contributions. First, we formally introduce our new data structure of template DDs as an extension of multi-valued DDs. They have dedicated box nodes which can themselves be identified with other template DDs, implementing box-associated decision making and leading to a hierarchical template DD structure. Second and third, we showcase the potential of template DDs by two applications:

1. Controller design with template DDs in the setting of *meta control*, and
2. synthesis of concise template DDs from DT controllers.

Here, we contribute to (1) with an example of modeling a control and meta control system with template DDs for the case study of an autonomous underwater vehicle. We show that through the separation of concerns in the control structures, modeling can be eased through template DDs by reusing decision making and control underspecification. Further, we contribute to (2) by refactoring DTs from standard dtControl benchmarks towards template DDs and hierarchical DDs. We implement a simple refactoring heuristics that already shows huge potential in template DDs for explainability by reducing control strategy representations by more than one order of magnitude.

2 Related Work

Decision Trees (DTs) and Decision Diagrams (DDs). DT classifiers have a long history in artificial intelligence (AI) and statistical learning [11,46]. The explanatory power of Boolean DTs recently gains more and more attention in explainable AI [6]. Plambeck *et al.* [50] presented an approach to connect environment input to system behaviors and thus foster understanding which input

caused what behavior. DTs are also used to generate *why*, *why-not*, *how-to*, and *what-if* explanations for context-aware intelligent systems [40].

While DTs enforce a tree structure, DDs are directed acyclic graphs and thus more general. Most prominently, binary DDs [12] are widely used in hardware design and verification to concisely represent Boolean functions [44,45]. The latter are in particular powerful when ordered and reduced, enabling a canonical representation and efficient operations. Extensions of binary DDs enable functions on more general co-domains (ADDs or MTBDDs, [26]), multi-valued decisions (multi-valued DDs, [58]), or nodes labeled by linear predicates instead of Boolean variables (linear DDs, [14]). MTBDDs for control strategy representation in synthesis have been considered in [63]. Recently, context-free DDs as hierarchical structures for efficient quantum circuit representation have been presented [57]. While this work also follows the idea of subdiagrams as components, advantages for modeling control strategies, explainability, and DT representations were not considered as we do with the use of template DDs.

Explainability of Self-adaptive Systems (SASs). Initially, research on SASs has been mainly focused on achieving self-adaptivity. However, due to increasing complexity of intelligent cyber-physical systems, explainability emerges as an important research direction [30,31]. Comprehensible adaptation rules for context-dependent effect expectations of adaptation options enable expressive, yet explainable, adaptation logics [36]. Li *et al.* proposed to use probabilistic model checking to determine when an explanation improves the overall system utility of an SAS with human-in-the-loop [39]. Camili *et al.* described different levels of explainability with corresponding meta-requirements, and demonstrated human-understandable explanations with robotic examples [13]. Autonomous agents can be also considered as adaptive systems. This research community has already a long tradition of investigating trust and explainability.

DTs have also been used as a specification formalism for SAS adaption rules, e.g., in the managing subsystem or meta controller for adapting control strategies [19,34,53]. Other works directly use DTs for self-adaptation: Chen *et al.* [15] model architectural design decisions available at runtime as DTs and combine them with goal models representing requirements for a new model-based self-adaptation approach. Cheng and Garlan [16] propose a language for describing adaptation strategies of architecture-based SASs where adaptation strategies are represented as DTs. None of these works uses a concept similar to template DDs.

Modularity in Decision Making. Behavior trees [20] are frequently used for robot controllers [33]. Their main advantage is a modular design where "individual behaviors can easily be reused in the context of another higher-level behavior, without needing to specify how they relate to the subsequent behaviors" [7]. There also exist approaches to parameterize behavior trees to improve reuse of subtrees [56] or reinforcement learning of parameters for complex robot control tasks [42]. Similar to these approaches, we introduce (parameterized) templates for DDs towards hierarchical template DDs that also benefit from modularity.

3 Foundations

We settle the foundations and notations used throughout the paper. Let us fix a set of *variables* Var on which we define *evaluations* as functions $\epsilon\colon \mathit{Var} \to \mathbb{Q}$ where $\mathbb{Q}$ are the rational numbers. The set of evaluations over Var is denoted by $\mathbb{E}$. A linear arithmetic *expression* e is defined by the grammar $e ::= c \mid c \cdot x \mid -e \mid (e + e)$ where x ranges over Var and c ranges over $\mathbb{Q}$. For an evaluation ϵ, the semantics $[\![e]\!]_\epsilon$ of an expression e is defined recursively as expected, e.g., $[\![x]\!]_\epsilon = \epsilon(x)$ and $[\![(e + e')]\!]_\epsilon = ([\![e]\!]_\epsilon + [\![e']\!]_\epsilon)$. We denote by $\mathit{Var}(e)$ the set of variables in the expression e. A linear *predicate* is of the form $(e \sim 0)$ where e is an expression and $\sim \in \{=, \geqslant, >\}$ is a comparison relation. The set of predicates is denoted by $\mathbb{P}$. A predicate $(e \sim 0) \in \mathbb{P}$ is *satisfied* by an evaluation ϵ, denoted $\epsilon \models (e \sim 0)$, iff $[\![e]\!]_\epsilon \sim 0$. We mainly consider *axis-aligned predicates*, i.e., predicates of the form $(c - x \geqslant 0)$ for $c \in \mathbb{Q}$ and $x \in \mathit{Var}$ and denote them by $x \leqslant c$. We also write x for the predicate $x - 1 \geqslant 0$, providing a Boolean interpretation of x.

Definition 1. *A* strategy *over states $S \subseteq \mathbb{E}$ and outcomes Out is a function $\sigma\colon S \to 2^{\mathit{Out}}$. We call σ* deterministic *if $|\sigma(\epsilon)| = 1$ for all $\epsilon \in S$. The strategy σ is* complete *if $S = \mathbb{E}$.*

To symbolically represent strategies through a decision diagram (DD) formalism that covers decision tree classifiers (DTs), we take inspiration from *linear DDs* [14] and *multi-terminal binary DDs* [26]. Recall that we fixed a set of variables Var used as domain of evaluations throughout the paper.

Definition 2. *A* decision diagram (DD) *is a tuple $\mathcal{D} = (N, \mathit{Out}, \lambda, \mathtt{lo}, \mathtt{hi}, r)$ where $N = D \cup O$ is a finite set of* nodes *comprising* decision nodes D *and* outcome nodes O *to which $\lambda\colon N \to \mathbb{P} \cup 2^{\mathit{Out}}$ labels either a predicate or a set of outcomes from Out, respectively, $\mathtt{lo}, \mathtt{hi}\colon D \to N$ are decision functions, and $r \in N$ is the root node of $\mathcal{D}$.*

A *path* in $\mathcal{D}$ is an alternating sequence $n_0, d_0, n_1, d_1, \ldots \in (D \times \{0,1\})^* \times N$ where for all i with $d_i = 1$ we have $n_{i+1} = \mathtt{hi}(n_i)$ and $n_{i+1} = \mathtt{lo}(n_i)$ if $d_i = 0$. Given a node $n \in N$, we denote by $\mathit{Reach}(n) \subseteq N$ the set of nodes reachable from n, i.e., the smallest set R where $n \in R$ and for all $n' \in R \cap D$ we have $\mathtt{lo}(n'), \mathtt{hi}(n') \in R$. We call $\mathcal{D}$ a *decision tree (DT)* if its underlying graph has a tree structure: for all $n, n' \in D$ with $n' \notin \mathit{Reach}(n)$ and $n \notin \mathit{Reach}(n')$ we have $\mathit{Reach}(n) \cap \mathit{Reach}(n') = \varnothing$. By $\mathcal{D}_n$ we denote the DD rooted at $n \in N$ that arises from $\mathcal{D}$ by replacing r by n. We assume that on all paths through $\mathcal{D}$, any predicate appears only once, i.e., $\lambda(n) = \lambda(n')$ implies $n = n'$ for all $n \in D$ and $n' \in \mathit{Reach}(n)$. We call $\mathcal{D}$ *deterministic* if $|\lambda(o)| = 1$ for all $o \in O$. Further, $\mathcal{D}$ is *reduced* if for all decision nodes $n, n' \in D$ we have $\mathtt{lo}(n) \neq \mathtt{hi}(n)$ (essentiality) and $\mathcal{D}_n \cong \mathcal{D}_{n'}$ implies $n = n'$ (merged isomorphic subdiagrams).

Figure 1 exemplifies how we graphically specify decision diagrams: the root node is specified by a sole incoming arrow; `hi`-edges are drawn with solid arrows, `lo`-edges with dotted ones; sets of outcomes are at the leafs of the diagram.

The semantics $[\![\mathcal{D}]\!]$ of $\mathcal{D} = \mathcal{D}_r$ is a complete strategy over $\mathbb{E}$ and Out, recursively defined for every evaluation $\epsilon \in \mathbb{E}$ as

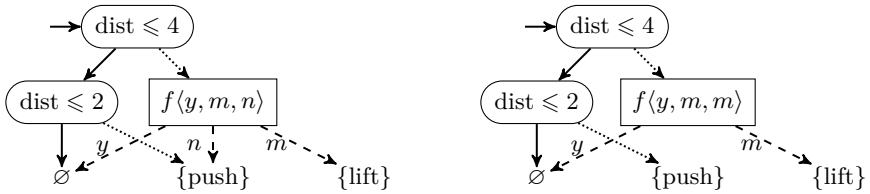

Fig. 4: Two simple example template DDs

$$[\![\mathcal{D}_n]\!](\epsilon) = \begin{cases} n & \text{if } n \subseteq \mathit{Out} \\ [\![\mathcal{D}_{\mathtt{lo}(n)}]\!](\epsilon) & \text{if } n \in D \text{ and } \epsilon \not\models \lambda(n) \\ [\![\mathcal{D}_{\mathtt{hi}(n)}]\!](\epsilon) & \text{if } n \in D \text{ and } \epsilon \models \lambda(n) \end{cases}$$

4 Template Decision Diagrams

In this section, we extend DDs towards *template DDs* where decisions are not exclusively made through completely specified predicates but can also be encapsulated in *boxes*. Formally, let $\mathbb{F}$ be a set of *function names* and define $\kappa(f)$ to denote the *arity* of the function $f \in \mathbb{F}$. Further, define a set $\mathbb{B}$ of *box types* that have the form $f\langle o_0, \ldots, o_{k-1}\rangle$ where $f \in \mathbb{F}$, $k = \kappa(f)$, and o_i for $0 \leqslant i < k$ are ordered *box outcomes*. The outcomes associated to a box type $\tau \in \mathbb{B}$, are collected in a set $\mathit{Out}(\tau) = \{o_0, \ldots, o_{k-1}\}$. Intuitively, a box of type τ as above encapsulates a strategy on $\mathit{Out}(\tau)$ by implementing function f. Enhancing DDs such that they can serve as templates of more complex decision making, we allow for *box nodes* that are non-binary as in *multi-valued DDs* [58]:

Definition 3. *A* template DD *is a tuple* $\mathcal{T} = (N, \mathit{Out}, \lambda, \mathtt{succ}, r)$ *where* $N = P \cup B \cup O$ *comprises* predicate nodes P, box nodes B, *and* outcome nodes O *to which* $\lambda\colon N \to \mathbb{P} \cup \mathbb{B} \cup 2^{\mathit{Out}}$ *labels either a predicate, a box type, or subsets from the set of outcomes Out, respectively. The decision function* succ *maps every predicate node* $p \in P$ *to a function* $\mathtt{succ}(p)\colon \{0,1\} \to N$ *and every box node* $b \in B$ *to a function* $\mathtt{succ}(b)\colon \mathit{Out}(\lambda(b)) \to N$, *and* $r \in N$ *is the root node of* $\mathcal{T}$.

Notations and definitions on DDs carry over to template DDs where we consider nodes $D = P \cup B$ as decision nodes. Similar to outcomes in box types, we assume template DD outcomes to be ordered, i.e., $\mathit{Out} = \{o_0, \ldots, o_{k-1}\}$ where $k = |\mathit{Out}|$. If a box type is Boolean, we omit the outcome types and assume o_0 standing for "false", "no", or "0", and o_1 standing for "true", "yes", or "1".

Figure 4 shows example two template DDs to specify a strategy that performs actions in $\mathit{Out} = \{\text{push}, \text{lift}\}$ depending on the distance "dist" to some object. The left template DD contains a box typed as $f\langle y, m, n\rangle$, providing three box outcomes y ("yes"), m ("maybe"), and n ("no"). Already this simple example shows one of the core benefits of template DDs, namely encapsulation and separation of concerns. Here, the decision whether to lift the object is externalized to

a strategy given by a function $f\langle y, m, n\rangle$ in case dist > 4. Otherwise, the decision to push iff dist > 2 is made. Similarly, the template DD on the right treats also the "no" outcome of function f as "maybe", leading to $Out(f\langle y, m, m\rangle) = \{y, m\}$ and hence a binary box.

Semantics of Template DDs. The definition of template DDs hints at decision making that is *underspecified*. For any box node $b \in B$ there is plenty of room which strategy $\sigma\colon \mathbb{E} \to 2^{Out(\lambda(b))}$ is actually represented by b. A natural approach to deal with such underspecification is to model outcomes nondeterministically, i.e., every box b represents a strategy σ where $\sigma(\epsilon) = Out(\lambda(b))$ for all evaluations $\epsilon \in \mathbb{E}$. Then, the semantics $[\![\mathcal{T}]\!]$ of a template DD $\mathcal{T} = \mathcal{T}_r$ as above is the complete strategy that is recursively defined over evaluations $\epsilon \in \mathbb{E}$:

$$[\![\mathcal{T}_n]\!](\epsilon) = \begin{cases} n & \text{if } n \subseteq Out \\ [\![\mathcal{T}_{\mathtt{succ}(n,0)}]\!](\epsilon) & \text{if } n \in P \text{ and } \epsilon \not\models \lambda(n) \\ [\![\mathcal{T}_{\mathtt{succ}(n,1)}]\!](\epsilon) & \text{if } n \in P \text{ and } \epsilon \models \lambda(n) \\ \bigcup_{o \in Out(\lambda(n))} [\![\mathcal{T}_{\mathtt{succ}(n,o)}]\!](\epsilon) & \text{if } n \in B \end{cases}$$

4.1 Hierarchical Decision Diagrams by Templates

Choosing box outcomes nondeterministically is usually too coarse when specifying strategies that have to fulfill certain requirements. A natural way to refine template DDs is by providing *function evaluations* that map boxes to more specific strategies over box outcomes. While our definitions allow for any representation of strategies, e.g., machine-learning classifiers, state-action lookup tables, etc., we especially have other template DDs as strategy representation for box evaluations in mind. Then, sets of template DDs provide a uniform formalism for strategy specification that follows a hierarchical structure similar to call graphs of procedural programs and context-free BDDs [57]:

Definition 4. *A* hierarchical DD *is a tuple* $\mathfrak{H} = (\mathcal{T}^0, \dots, \mathcal{T}^{m-1}, \beta)$ *where* $\mathcal{T}^i = (N^i, Out^i, \lambda^i, \mathtt{succ}^i, r^i)$ *are template DDs for* $i \in \{0, \dots, m-1\}$ *and* $\beta\colon \mathbb{F} \to \{0, \dots, m-1\}$ *is a* function evaluation *that maps function names to template DDs such that for all* $f \in \mathbb{F}$*:*

- $\mathcal{T}^{\beta(f)}$ *is compatible with* f*, i.e.,* $|Out^{\beta(f)}| = \kappa(f)$*, and*
- $\mathcal{T}^{\beta(f)}$ *is decreasing, i.e.,* $\beta(\lambda^{\beta(f)}(b)) > \beta(f)$ *for all* $b \in B^{\beta(f)}$*.*

The latter two conditions on function evaluations ensure that evaluated boxes provide a decision on outcomes and that no recursion occurs, respectively. This leads to a well-defined semantics $[\![\mathfrak{H}]\!] = [\![\mathcal{T}^0_{r^0}]\!]_\beta$ of a hierarchical DD $\mathfrak{H}$ with function evaluation β, recursively defined over evaluations $\epsilon \in \mathbb{E}$:

$$[\![\mathcal{T}_n^i]\!]_\beta(\epsilon) = \begin{cases} n & \text{if } n \subseteq Out^i \\ [\![\mathcal{T}^i_{\mathtt{succ}^i(n,0)}]\!]_\beta(\epsilon) & \text{if } n \in P^i \text{ and } \epsilon \not\models \lambda^i(n) \\ [\![\mathcal{T}^i_{\mathtt{succ}^i(n,1)}]\!]_\beta(\epsilon) & \text{if } n \in P^i \text{ and } \epsilon \models \lambda^i(n) \\ \bigcup_{k<\kappa(f):o_k^j \in [\![\mathcal{T}^j_{r^j}]\!]_\beta(\epsilon)} [\![\mathcal{T}^i_{\mathtt{succ}^i(n,o_k)}]\!]_\beta(\epsilon) & \text{if } n \in B^i \text{ with } j{=}\beta(f) \text{ and} \\ & \lambda^i(n) = f\langle o_0, \ldots, o_{\kappa(f)-1}\rangle. \end{cases}$$

The last case of the above case distinction formalizes the evaluation of the jth template DD at its root r^j invoked from a box n in the ith template DD. For each outcome o_k^j the kth outcome o_k of the box type in the ith template DD is then chosen, continuing the evaluation within the ith template DD.

4.2 Standard Template Boxes

Templating through boxes can be done to specify common decision patterns that often occur in decision strategies or to encapsulate a sub-decision for reuse in a strategy-representing DD. The corresponding concept of procedures known from software engineering also provides inspiration for common structures how boxes are instantiated in most common applications.

Templates with Parameters. Often, recurring sub-decisions have the same structure but can differ in the values of constants used in the predicates of decision nodes. A common way to deal with that in software engineering is to introduce parameters. To enable the use of *parameterized templates*, we propose to introduce an additional variable in box types that serves as a parameter. For readability, we propose to list parameter names in square brackets behind the function name, e.g., parametrizing a box type $f\langle o_0, \ldots, o_{k-1}\rangle$ with parameters p_0 to p_{n-1} yields

$$f[p_0, \ldots, p_{n-1}]\langle o_0, \ldots, o_{k-1}\rangle$$

Standard Templates. In control decisions, we often have the same kind of checks that are performed on observable system variables. To provide explicit modeling solutions for these well-known patterns, we introduce standard templates and feasible implementations by (template) DDs. Next to simple checks like *less*, *equal*, or *greater*, variables often have to be within a certain interval and control actions depend on whether the variable is *smaller*, *within*, or *greater* than interval boundaries. We provide two templates for interval containment. First, templates with the box type "in_interval2$[x, \ell, u]\langle \text{in}, \text{out}\rangle$" decide whether a variable x is evaluated within the interval $[\ell, u]$ or outside. It corresponds to the DD depicted in Figure 5. This check is often used in control decisions to decide whether an action needs to be performed in order to keep the process variable within a reference interval. Second, control decision may also depend on whether the value is below or above the desired range. For this, we provide a template with box type "in_interval3$[x, \ell, u]\langle \text{in}, <, > \rangle$", which decides whether a variable x is evaluated within the interval $[\ell, u]$, smaller, or greater than the interval. The corresponding DT is depicted in Figure 6.

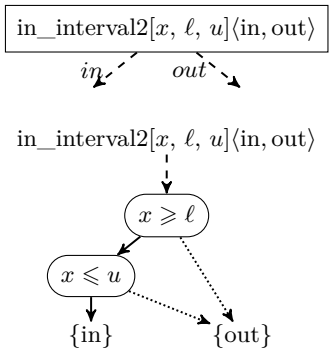

Fig. 5: Interval2 template (top) and a feasible implementation (bottom)

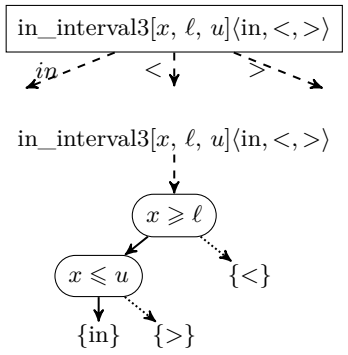

Fig. 6: Interval3 template (top) and a feasible implementation (bottom)

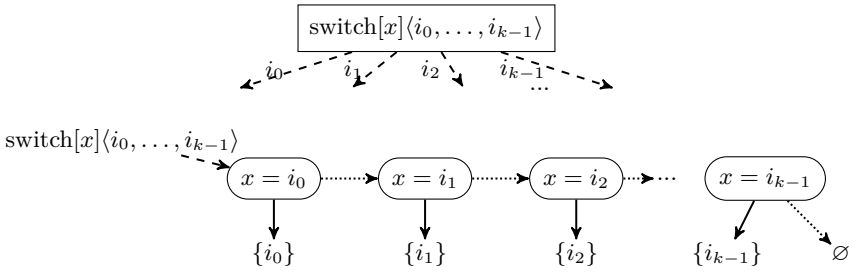

Fig. 7: Switch template (top) and a feasible implementation (bottom)

Another recurring pattern are *switches*, that compare the variable to several values and execute a corresponding control decision afterwards. The template of box type "switch$[x]\langle i_0, \ldots, i_{k-1}\rangle$" decides whether variable x is evaluated to i_0, i_1, ..., or i_{k-1}. This pattern is used in programming languages to avoid nested if-statements. The corresponding DT is depicted in Figure 7. Note that also alternative implementations of this template could be imagined, e.g., using binary search, from which one can abstract by using templates and leaving implementation details to future development steps or when further explanations on the implementation are required.

5 Templates for Self-Adaptive Systems and Meta Control

Decision trees (DTs) as an instance of DDs are well-established to specify control policies. Using template DDs instead of DDs further provides two main advantages: (i) it accommodates reuse of decision policies and thereby introduces a separation of concerns by enclosing common behaviors into boxes, similar to procedures in programming, and (ii) it allows underspecification of certain parts of the control strategy, thereby enabling specification of families of controllers.

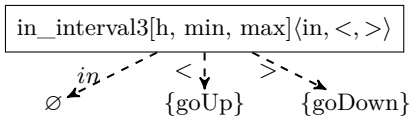

Fig. 8: Example template DD with standard interval template for an AUV

We illustrate the use of template DDs as modeling formalism for control and meta control policies with template DDs. As explained in Section 1, an SAS focussed on adapting control strategies can be seen as a special case of a two-layered system with a managed and a managing subsystem (see Figure 2) where the managed subsystem is a control system and the managing subsystem is a meta control system. The meta control system adapts the control strategies of the control system according to input from the environment and the control system. The control system can be specified as a template DD, making common decision strategies explicit and thereby making it easier to understand the control system. Furthermore, template DDs can be left underspecified and contain parameters which are encoded as variables in the function name.

By instantiating template DDs and parameters of functions instead of adapting the control strategy of the control system in a different way, the adaptation strategy of the meta control system (i) is easier to specify because changing parameters of functions gives a good overview of the choices that can be taken, and the structure of the template DDs can be reused, (ii) has a compact structure which makes it easier to understand. The meta control system can itself be represented as a template DD.

We illustrate how interacting control and meta control systems can be modeled with template DDs on a case study inspired by the exemplar SUAVE [52]. To do so, the case study is first described in Section 5.1 and then modeled with template DDs in Section 5.2. As a first example for how to represent template DDs in a control system, Figure 8 shows a template DD for an autonomous underwater vehicle (AUV). The template DD makes use of our standard template for interval checking, which can be implemented as shown in Section 4.2. When considering the AUV to be a two-layered system, the instantiation of the function parameters would be done by the meta control system, setting the values for *min* and *max* to adapt the minimal and maximal diving depth of the AUV.

5.1 An Overview of the Case Study

In this section, we introduce our case study of an AUV used for searching for a pipeline located on a seabed and then following and inspecting it. The AUV has two kinds of vision sensors, a camera and a sonar, where the sonar consists of a side-scan sonar and a forward-facing sonar. Both kinds of vision sensors can be used for searching for the pipeline and for following it, but the sonar is preferred for searching since it can operate on a higher altitude, giving a bigger field of view, and the camera is preferred for inspecting the pipeline since it is easier to detect faults of the pipeline with the camera.

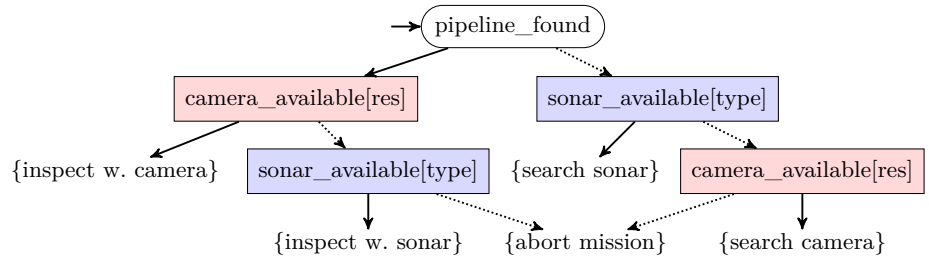

Fig. 9: The control system of the case study as a template DD

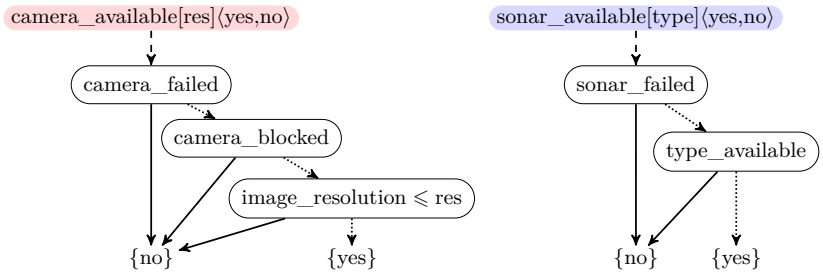

Fig. 10: The template DD instances for the control system

Depending on the current task of the AUV, i.e., searching for the pipeline or following it, a different camera resolution or sonar is required. When searching for the pipeline with the camera, the resolution does not need to be as high as for inspection since during inspection, faults need to be identified with the camera images. When searching for the pipeline with the sonar, the side-scan sonar is preferred since it provides a wider field of view than the forward-facing sonar. However, when following the pipeline, the forward-facing sonar is preferred. Both the sonar and the camera might fail during runtime, making it necessary to choose the other vision sensor for both tasks. Furthermore, the camera can get blocked temporarily, making a temporary switch to the sonar necessary for inspection. If both sensors fail, the mission should be aborted.

5.2 Modeling the Case Study with Template DDs

The AUV can be modeled as an SAS with a control system responsible for searching for and inspecting the pipeline, and a meta control system that adapts the control strategy of the control system depending on the current task of the AUV. Therefore, we model the control system of the AUV as a hierarchical template DD (see Figures 9 and 10), and the meta control system as a (template) DD without box nodes that determines the values for the control system's function parameters depending on the state of the control system (see Figure 11).

Control System. The control system consists of the hierarchical template DD

representing how to choose the sensor for searching for the pipeline and following it, see Figure 9. When searching for the pipeline (the pipeline has not been found, indicated by the Boolean value *pipeline_found*), the sonar is the preferred vision sensor so it is chosen if available. If only the camera is available, then the camera is chosen. Otherwise the mission should be aborted. When the pipeline has been found, the camera is preferred for following it and the sonar is only chosen if the camera is not available but the sonar is. In both cases, if the pipeline has been found and if it has not, the control system has to determine whether the camera and the sonar are available. These checks depend on the parameters res, the "resolution" of the camera, and type, the type of sonar used (side-scan and forward-facing), and have been summarized in the box types "camera_available[res]" and "sonar_available[type]", respectively. Recall that we simplify box types and outgoing edges of box types if they have only two outcomes, i.e., the box types "camera_available[res]$\langle o_0, o_1\rangle$" and "sonar_available[type]$\langle o_0, o_1\rangle$" have been simplified by omitting the box outcomes o_0 and o_1, and the outgoing edges of the box types are drawn as usual "yes/no-edges" for simplification.

Using the box types for checking if the camera and sonar are available makes understanding and adapting the control strategy of the control system much easier because (i) the box types summarize (template) DDs, increasing explainability and clarifying the structure of the DD; without them, this structure would be difficult to spot, (ii) the box nodes can be reordered easily to reflect a different preference for using camera and sonar; with the preferences described in Section 5.1, the two box types appear in a different order in the sub-template DDs, depending on whether the pipeline has been found.

The camera and sonar have to be available with different resolution and type, respectively, depending on the current task of the AUV, see Section 5.1. Therefore, the chosen instantiations of these box types depend on the parameters *res* for "resolution" and type for the type of sonar to be used (side-scan or forward-facing). The template DDs implementing them are depicted in Figure 10, where "camera_failed", "camera_blocked", "sonar_failed", and "type_available" are Boolean variables that are assumed to be set to the correct value by the system depending on sensor input on whether the camera failed, the camera is blocked, the sonar failed and the requested type is available, respectively. The variable "image_resolution" is an integer that is also set by the system depending on the current resolution of the camera images. In conclusion, the control system of the AUV is a hierarchical DD $\mathfrak{H} = (\mathcal{T}^0, \mathcal{T}^1, \mathcal{T}^2, \beta)$ where $\mathcal{T}^0, \mathcal{T}^1$, and $\mathcal{T}^2$ are the template DDs for the control system (Figure 9), the camera and sonar available DDs (Figure 10), respectively, and

$$\beta\colon \{\text{control}, \text{camera\_available[res]}, \text{sonar\_available[type]}\} \to \{0, 1, 2\}$$

$$\beta(f) = \begin{cases} 0 & \text{if } f = \text{control}, \\ 1 & \text{if } f = \text{camera\_available[res]}, \text{ and} \\ 2 & \text{if } f = \text{sonar\_available[type]}. \end{cases}$$

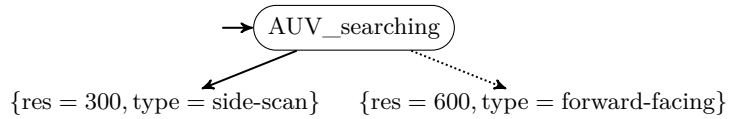

Fig. 11: The meta-control system of the case study

Meta Control System. The meta control system adapts the control strategy of the control system by instantiating the parameters res and type of the functions "camera_available[res]" and "sonar_available[type]" of the control system depending on whether the AUV is searching for the pipeline or following it, see Figure 11. The variable "AUV_searching" is a Boolean variable that is assumed to be set by the system depending on the state of the control system, i.e., depending on whether the AUV is searching for the pipeline. This variable is different from the variable "pipeline_found" in the control system which captures whether the pipeline has been recognized on the sensor input.

Enabling adaptation of the control system by instantiating parameters of functions enables reuse of box types and makes it easy to spot which parameters can be adapted and where they are used. With box types, the control strategy of the control system can easily be understood on a high level (i.e., without having to look at the implementation of the camera and sonar check) and the adaptation strategy of the meta control system is easy to understand and represent. Without box types, it would both be harder to see which parameters can be adapted and how the parameters in different parts of the DD relate to each other.

6 Improving Control Strategy Explanations

In software engineering, well-established concepts help to improve understandability and maintainability of software: separation of concerns, encapsulation, and reuse of behaviors. Their common intend is to reduce the amount of code that has to be captured and understood during software development. By encapsulating decisions and actions that belong together into subprograms, objects, and procedures, the overall program can be understood on a more abstract level. Encapsulated functionalities then only need to be evaluated once when inspecting software. Further, when changing functionalities or debugging, code modifications are reduced and mitigate negative side-effects of code duplication.

Our definition of *template DDs* and hence *hierarchical DDs* has been inspired by procedural software and thus can benefit from these concepts towards improved explainability of DTs and DDs. As boxes in template DDs usually encapsulate complex behavior including several decisions, a template DD is smaller than the equivalent DD without boxes, and, thus, easier to grasp. Furthermore, boxes allow for separation of concerns and can be examined individually.

6.1 Explainability Metrics for Template DDs

While these arguments are purely qualitative, the software engineering community also provides metrics that allow to quantify the understandability of code [64]. In the following, we describe the most important ones and transfer them to our template DD setting. Following Occam's razor, the best explanation for a phenomenon is the smallest one amongst all possible explanations [28]. Metrics to measure the psychological complexity [64] and hence understandability of source code can hence be used as metrics for explainability. Minimizing such metrics corresponds to increasing explainability. Most prominently used are the number of *lines of code (LOC)* and McCabe's *cyclomatic complexity (CC)* [43,61]. The latter measures the number of linearly independent control flows through a program. Formally, CC is defined as the sum of $|E| - |N| + 2$ for each procedure, where $|E|$ and $|N|$ are the number of edges and nodes of the control-flow graph, respectively. Note that with binary decisions and one terminal node only, this measure simplifies to $|N|$, i.e., the LOC measure.

Let us now establish corresponding metrics to measure the complexity of template DDs. For this, let $\mathcal{T} = (N, Out, \lambda, \texttt{succ}, r)$ be a template DD. The LOC measure on source code can directly be transferred to the setting of DDs as the number of decision nodes. This allows for a first impression of the size of information that needs to be understood. Formally,

$$LOC(\mathcal{T}) \;=\; |N|.$$

McCabe's cyclomatic complexity can also be interpreted on template DDs, where the DD is seen as control-flow graph in which each box corresponds to a procedure call and where each terminal node leads to an artificial control-flow sink node. Formally, we define the CC measure as

$$CC(\mathcal{T}) = |P| + \sum\nolimits_{b \in B} \big(|\mathit{Out}(\lambda(b))| - 1\big) + 1.$$

Note that each predicate node has two successors, while boxes might have higher arity. Both measures naturally extend to hierarchical DDs $\mathfrak{H} = (\mathcal{T}^0, \ldots, \mathcal{T}^{m-1}, \beta)$ as the sum of local measures, i.e.,

$$LOC(\mathfrak{H}) \;=\; \sum\nolimits_{i=0}^{m-1} LOC(\mathcal{T}^i) \quad \text{and} \quad CC(\mathfrak{H}) \;=\; \sum\nolimits_{i=0}^{m-1} CC(\mathcal{T}^i).$$

In the hierarchical DD of the Triangle control example (see Figure 3), we obtain $CC(\mathcal{T}^0) = 7 + 2 \cdot 3 + 1 = 14$ and $CC(\mathcal{T}^1) = 3 + 1 = 4$, leading to a cyclomatic complexity of $CC(\mathfrak{H}) = 18$. Note that $CC(\mathcal{D}) = 13 + 0 + 2 = 14$ in the DT representation of the very same control strategy (see Figure 1). With respect to the LOC measure, we achieve better explainability for this example by $LOC(\mathfrak{H}) = 25$ compared to $LOC(\mathcal{D}) = 27$.

Note that the overall cyclomatic complexity $CC(\mathfrak{H})$ increased with refactoring due to the overhead of a second diagram where profit from reuse is not mitigated by the small size of the extracted subdiagram. However, when considering the meta control example in Figure 9 and Figure 10, we get $CC(\mathcal{D}) = 12$

for a non-refactored AUV and $CC(\mathcal{T}^0) = 6$, $CC(\mathcal{T}^1) = 4$ and $CC(\mathcal{T}^2) = 3$. Here, the overall cyclomatic complexity $CC(\mathfrak{H}) = 6 + 4 + 3 = 13$ is also slightly higher than $CC(\mathcal{D}) = 12$ but the individual diagrams that can be investigated in isolation have a much smaller cyclomatic complexity and have the benefit of explicitly showing code reuse.

6.2 Decision Diagram Refactoring

In Section 5, we showcased how template DDs can facilitate modeling of control systems with reuse of decision making. Legacy monolithic control systems, however, do not benefit from the opportunities of concise representation and reuse yet. *Refactoring* [25] is the standard approach in software engineering to improve the design of legacy code by encapsulating functionalities. This concept can also be used in the setting of DDs, factoring out isomorphic subdiagrams into template DDs towards hierarchical DDs that improve explainability measures.

Decision Diagram Refactoring Problem: Given a template DD $\mathcal{T}$ and an explainability measure η, synthesize a hierarchical DD $\mathfrak{H}_{\mathcal{T}} = (\mathcal{T}^0, \dots, \mathcal{T}^{k-1}, \beta)$ such that $[\![\mathfrak{H}_{\mathcal{T}}]\!] = [\![\mathcal{T}]\!]$, $\bigcup_{i=0}^{k-1} P^i \subseteq P$, and $\eta(\mathfrak{H}_{\mathcal{T}})$ is minimal.

Note that we required the predicate nodes of the input template DD $\mathcal{T}$ to cover the predicate nodes of all template DDs $\mathcal{T}^i$ in the synthesized hierarchical DD $\mathfrak{H}_{\mathcal{T}}$. Intuitively, this corresponds to reusing nodes of $\mathcal{T}$ in template DDs and removing duplicates. Solving the refactoring problem clearly is intractable, since finding minimal refactoring involves to solve an NP-hard problem. To this end, heuristics for refactoring are favorable towards improving explainability of DD strategies. A naive greedy algorithm on a hierarchical DD $\mathfrak{H} = (\mathcal{T}^0, \dots, \mathcal{T}^{k-1}, \beta)$ over predicate nodes $P = \bigcup_{i=0}^{k-1} P^i$ to factor out one promising template DD $\mathcal{T}^k$ would be to compute the maximal gain of explainability for isomorphic subdiagrams and opt for factoring out the maximal one:

1. Initialize the maximal gain $g_{\max} := 0$ and a template candidate $\mathcal{F} := \varnothing$.
2. For each pair $n \in P^i$ and $m \in P^j$ with $\lambda^i(n) = \lambda^j(m)$
   (a) determine the maximal subdiagram $\mathcal{S}$ of $\mathcal{T}^i_n$ such that
      – $\mathcal{S}$ is isomorphic to the top subdiagram of $\mathcal{T}^j_m$, and
      – $\mathcal{S}$ does not contain a node with multiple incoming edges;
   (b) compute the gain $g = \eta(\mathfrak{H}) - \eta(\mathfrak{H}')$ where $\mathfrak{H}'$ arises from $\mathfrak{H}$ by replacing every isomorphic occurrence of $\mathcal{S}$ in $\mathfrak{H}$ by a box pointing to a tentative fresh template DD $\mathcal{T}^k := \mathcal{S}$;
   (c) if $g > g_{\max}$, set $g_{\max} := g$ and set $\mathcal{F} := \mathcal{S}$.
3. Update $\mathfrak{H}$ with a fresh template DD $\mathcal{T}^k := \mathcal{F}$ and replace every isomorphic occurrence of $\mathcal{F}$ in $\mathfrak{H}$ by a box pointing to $\mathcal{T}^k$.

The above procedure can be initially applied to a hierarchical DD comprising a single DD only, and repeated until no gain in explainability is achieved anymore. Note that factoring out a template DD introduces new outcome nodes in the freshly generated template component, increasing explainability measures if there are only few template occurrences or small template sizes.

6.3 Implementation and Evaluation

We implemented template DDs and support for hierarchical DDs building on the Python-based DT learning framework dtControl [5]. For this, we first transfer DTs learned with dtControl into reduced DDs by merging outcome nodes and isomorphic subtrees. Then, we apply our refactoring heuristics from the last section and evaluate the LOC explainability measure, i.e., improving explainability through reducing DD sizes. Template DDs can be exported graphically towards explainable control strategies – Figures 1 and 3 are generated by our implementation. Our implementation and experiments are publicly available[4].

Experiments. To investigate the explanatory potential of hierarchical DDs, we applied DD generation and refactoring through our heuristics to the standard benchmark set of dtControl, comprising different control strategies learned from cyber-physical systems, classical planning tasks, machine learning, as well as synthesized by probabilistic model checkers from examples of the Prism benchmark suite [38]. Note that for DD generation, we only applied reduction rules to the DT output, not changing the order of predicates. As explainability metrics we chose the LOC measure, since CC does not account well for reuse in templates (see Section 6.1). Table 1 shows the stepwise node-count reductions, where in the first half standard Prism benchmarks and in the second half cyber-physical systems benchmarks are separated through a horizontal line. The "all" node counts are reflecting the standard LOC measure, while "inner" refers to the number of inner nodes, i.e., predicate and box nodes. The latter is more meaningful to mitigate the clear drawback of DTs in multiple outcome node duplications. Thus, we also provide reductions only w.r.t. inner nodes, leading to an inner reduction of $1 - \frac{\textbf{factor(DD)}}{\textbf{DT}}$. Here, **DT** is the number of inner DT nodes and **factor(DD)** stands for applying our refactoring heuristic to the DD obtained from the original DT input from dtControl. We can observe clear reductions in size, even more than one order of magnitude in the case of the "10rooms" benchmark from the cyber-physical systems domain. There are diverse reasons for reductions, which can be explained by the diverse benchmark set. Reductions are both, due to switching from DTs to DDs as representation formalism, and refactoring to hierarchical DDs. While refactoring has great impact in almost all cases, switching to DD formalism sometimes does not provide any reduction of inner nodes. For instance, in the case of the "cdrive" benchmark, refactoring provides the only source of more than 60% reduction.

7 Conclusion

We introduced templates in decision diagrams (DDs) as an extension of multi-valued DDs, which allow to encapsulate and reuse common decision-making patterns. Therefore, template DDs can yield more explainable representations than standard decision trees (DTs) due to their smaller size and by introducing

[4] `https://anonymous.4open.science/r/tdd-97EC`

Table 1: Results for dtControl experiments

	DT		DD		factor(DD)		inner
Model	**all**	**inner**	**all**	**inner**	**all**	**inner**	**reduction**
beb	65	32	61	32	62	30	6%
blocksworld	1,235	617	800	616	855	572	7%
cdrive	2,401	1,200	2,152	1,200	2,049	422	65%
consensus	67	33	45	32	51	29	12%
csma	103	51	83	50	86	49	4%
eajs	167	83	112	79	115	78	6%
echoring	1,869	934	1,267	866	1,329	790	15%
elevators	16,327	8,163	6,325	6,260	4,696	4,348	47%
exploding-blocksworld	4,981	2,490	2,405	2,330	2,344	1,907	23%
firewire	307	153	232	153	237	132	14%
ij	1,291	645	510	500	525	418	35%
leader4	125	62	67	55	75	47	24%
mer30	137	68	89	68	91	61	10%
pacman	43	21	40	21	40	21	0%
philosophers-mdp	391	195	222	192	228	170	13%
pnueli-zuck	171,371	85,685	69,637	69,550	55,605	49,050	43%
rabin	111	55	67	55	70	47	15%
rectangle-tireworld	481	240	481	240	435	155	35%
triangle-tireworld	27	13	22	13	25	12	8%
wlan_dl	3,369	1,684	1,613	1,525	1,680	1,400	17%
zeroconf	381	190	187	178	197	165	13%
10rooms	17,297	8,648	748	723	534	313	96%
aircraft	915,877	457,938	332,424	332,408	307,395	264,419	42%
cartpole	253	126	211	126	218	116	8%
dcdc	271	135	138	135	138	135	0%
helicopter	6,339	3,169	3,079	2,841	3,049	2,312	27%
traffic_30m	12,573	6,286	2,888	2,876	2,950	2,281	64%
truck_trailer	338,283	169,141	142,915	141,995	138,245	111,975	34%
vehicle	13,235	6,617	5,878	5,860	6,261	5,256	21%

different levels of abstraction in a hierarchical template DD. Furthermore, they enable underspecification which can be exploited for modeling meta control. We illustrated the benefits of template DDs for modeling and explaining meta controllers with a case study from the autonomous underwater robotics domain. To evaluate the potential benefits of template DDs for existing controllers given as DTs, we implemented a template-generating refactoring method for DTs. Our results on standard controller benchmarks showed template DDs to improve explainability of by reducing DT sizes by more than one order of magnitude.

Our new data structure for explainable AI and control strategy representation can be seen as a starting point for several future investigations. First, we did not opt for *ordered* template DDs, a common requirement usually present in binary DDs [12]. While this could lead to performance advantages for operations on DDs and allows for integration in state-of-the-art DD frameworks [32], this restriction would limit the modeling power: permuting templates as we illustrated in Figure 9 would be not contained in the class of ordered template DDs. However, integration of template DDs and the impact of the ordering condition are an interesting future direction. Second, template DDs could be used as runtime models in self-explainable systems that autonomously decide when and how to explain themselves. Explanations could here be supported by what-if analyses for possible template instantiations or meta control decisions, importance of decisions, and causal reasoning [8,29,23]. Self-explanation can be also achieved by adding a meta-layer, similar to SASs, that monitors the system and constructs explanations from an abstract model of the system [22,10,24]. In [55], we proposed a high-level process to extract tree-like explanation models from system models with timing information. Here, template DDs could ease extraction and enable explanations at different levels of abstraction. Also on-the-fly adaptations of parameters and template instantiations could improve control performance and explainability, where in-depth considerations are future work. Third, learning algorithms that directly provide template DDs instead of DTs for control strategies could even further increase sharing and reuse of decision making, directly impacting existing methods in explainable AI. Our refactoring experiments conclusively demonstrated the potential of possible improvements and provide a lower bound on how explainability could be increased with dedicated learning algorithms. In this vein, also the detection of appearances of standard templates as presented in Section 4.2 could be investigated, e.g., factoring out switch chains common in outputs from dtControl strategies. Fourth, the concept of templates in DD could be further investigated with applications in control theory [4] and reinforcement learning by templating and bootstrapping parts of the learning process [59,37,48,9]. Last but not least, further explainability measures accompanied with user studies to investigate the human understanding of control strategy explanations by template DDs would be important to achieve a profound integration of template DDs into engineering processes.

Acknowledgments. This work was partially supported by the DFG under the projects EXC 2050/1 (CeTI, project ID 390696704, as part of Germany's Ex-

cellence Strategy) and TRR 248 (see https://perspicuous-computing.science, project ID 389792660), by the NWO through Veni grant VI.Veni.222.431, and by the European Union's Horizon 2020 Framework Programme through the MSCA network REMARO (Grant Agreement No 956200). Authors in alphabetic order.

References

1. Ethics guidelines for trustworthy AI - european commission, directorate-general for communications networks, content and technology (2019), https://data.europa.eu/doi/10.2759/177365
2. Four principles of explainable artificial intelligence - (U.S.) national institute of standards and technology (NIST) (2020), https://doi.org/10.6028/NIST.IR.8312-draft
3. Proposal for a regulation laying down harmonised rules on artificial intelligence (artificial intelligence act) and amending certain union legislative acts, com(2021) 206 final - european commission (2021), https://ec.europa.eu/transparency/regdoc/rep/1/2021/EN/COM-2021-206-F1-EN-MAIN-PART-1.PDF
4. Anand, A., Nayak, S.P., Schmuck, A.K.: Synthesizing permissive winning strategy templates for parity games. In: Enea, C., Lal, A. (eds.) Computer Aided Verification. pp. 436–458. Springer Nature Switzerland, Cham (2023)
5. Ashok, P., Jackermeier, M., Kretínský, J., Weinhuber, C., Weininger, M., Yadav, M.: dtcontrol 2.0: Explainable strategy representation via decision tree learning steered by experts. In: TACAS (2). Lecture Notes in Computer Science, vol. 12652, pp. 326–345. Springer (2021)
6. Audemard, G., Bellart, S., Bounia, L., Koriche, F., Lagniez, J.M., Marquis, P.: On the explanatory power of boolean decision trees. Data & Knowledge Engineering **142**, 102088 (2022). https://doi.org/https://doi.org/10.1016/j.datak.2022.102088
7. Bagnell, J.A., Cavalcanti, F., Cui, L., Galluzzo, T., Hebert, M., Kazemi, M., Klingensmith, M., Libby, J., Liu, T.Y., Pollard, N.S., Pivtoraiko, M., Valois, J., Zhu, R.: An integrated system for autonomous robotics manipulation. In: 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2012, Vilamoura, Algarve, Portugal, October 7-12, 2012. pp. 2955–2962. IEEE (2012). https://doi.org/10.1109/IROS.2012.6385888
8. Baier, C., Dubslaff, C., Funke, F., Jantsch, S., Majumdar, R., Piribauer, J., Ziemek, R.: From verification to causality-based explications. In: Proceedings of the 48th International Colloquium on Automata, Languages, and Programming (ICALP) (2021)
9. Baier, C., Dubslaff, C., Wienhöft, P., Kiebel, S.J.: Strategy synthesis in markov decision processes under limited sampling access. In: Rozier, K.Y., Chaudhuri, S. (eds.) NASA Formal Methods. pp. 86–103. Springer Nature Switzerland, Cham (2023)
10. Blumreiter, M., Greenyer, J., Garcia, F.J.C., Klös, V., Schwammberger, M., Sommer, C., Vogelsang, A., Wortmann, A.: Towards self-explainable cyber-physical systems. In: 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C). pp. 543–548 (2019)
11. Breiman, L., Friedman, J.H., Olshen, R.A., Stone, C.J.: Classification and Regression Trees. Wadsworth (1984)

12. Bryant, R.E.: Symbolic Boolean manipulation with ordered binary-decision diagrams. ACM Computing Surveys **24**(3), 293–318 (Sep 1992). https://doi.org/10.1145/136035.136043
13. Camilli, M., Mirandola, R., Scandurra, P.: Xsa: explainable self-adaptation. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. pp. 1–5 (2022)
14. Chaki, S., Gurfinkel, A., Strichman, O.: Decision diagrams for linear arithmetic. In: 2009 Formal Methods in Computer-Aided Design. pp. 53–60 (2009). https://doi.org/10.1109/FMCAD.2009.5351143
15. Chen, B., Peng, X., Yu, Y., Nuseibeh, B., Zhao, W.: Self-adaptation through incremental generative model transformations at runtime. In: Jalote, P., Briand, L.C., van der Hoek, A. (eds.) 36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014. pp. 676–687. ACM (2014). https://doi.org/10.1145/2568225.2568310
16. Cheng, S., Garlan, D.: Stitch: A language for architecture-based self-adaptation. J. Syst. Softw. **85**(12), 2860–2875 (2012). https://doi.org/10.1016/J.JSS.2012.02.060
17. Chrszon, P., Baier, C., Dubslaff, C., Klüppelholz, S.: Interaction detection in configurable systems – a formal approach featuring roles. Journal of Systems and Software **196**, 111556 (2023)
18. Chrszon, P., Dubslaff, C., Klüppelholz, S., Baier, C.: Profeat: feature-oriented engineering for family-based probabilistic model checking. Formal Aspects of Computing **30**(1), 45–75 (2018)
19. Cioara, T., Anghel, I., Salomie, I., Dinsoreanu, M., Copil, G., Moldovan, D.: A self-adapting algorithm for context aware systems. In: 9th RoEduNet IEEE International Conference. pp. 374–379 (2010)
20. Colledanchise, M., Ögren, P.: Behavior trees in robotics and AI: an introduction. CoRR **abs/1709.00084** (2017)
21. Dijkstra, E.W.: On the role of scientific thought. In: Selected Writings on Computing: A Personal Perspective, pp. 60–66. Springer, Berlin (1982), transcribed 1974
22. Drechsler, R., Lüth, C., Fey, G., Güneysu, T.: Towards self-explaining digital systems: A design methodology for the next generation. In: 3rd International Verification and Security Workshop (IVSW). pp. 1–6. IEEE (2018). https://doi.org/10.1109/IVSW.2018.8494900
23. Dubslaff, C., Weis, K., Baier, C., Apel, S.: Feature causality. Journal of Systems and Software **209**, 111915 (2024). https://doi.org/https://doi.org/10.1016/j.jss.2023.111915
24. Fey, G., Fränzle, M., Drechsler, R.: Self-explanation in systems of systems. In: 2022 IEEE 30th International Requirements Engineering Conference Workshops (REW). pp. 85–91. IEEE (2022)
25. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston, MA, USA (1999)
26. Fujita, M., McGeer, P.C., Yang, J.C.: Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. Formal Methods Syst. Des. **10**(2/3), 149–169 (1997)
27. Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. In: POPL. pp. 499–512. ACM (2016)
28. Good, I.J.: Explicativity: a mathematical theory of explanation with statistical applications. Proc. R. Soc. Lond. **A354**, 303–330 (1977)

29. Harder, H., Jantsch, S., Baier, C., Dubslaff, C.: A unifying formal approach to importance values in boolean functions. In: Elkind, E. (ed.) Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI-23. pp. 2728–2737. International Joint Conferences on Artificial Intelligence Organization (8 2023). https://doi.org/10.24963/ijcai.2023/304, main Track
30. Hezavehi, S.M., Weyns, D., Avgeriou, P., Calinescu, R., Mirandola, R., Perez-Palacin, D.: Uncertainty in self-adaptive systems: A research community perspective. ACM Trans. Auton. Adapt. Syst. **15**(4) (2021). https://doi.org/10.1145/3487921
31. Horváth, I., Tavčar, J.: Designing cyber-physical systems for runtime self-adaptation: Knowing more about what we miss... Journal of Integrated Design and Process Science **25**(2), 1–26 (2021). https://doi.org/DOI10.3233/JID210030
32. Husung, N., Dubslaff, C., Hermanns, H., Köhl, M.A.: OxiDD: A safe, concurrent, modular, and performant decision diagram framework in Rust. In: Finkbeiner, B., Kovács, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 255–275. Springer Nature Switzerland, Cham (2024)
33. Iovino, M., Scukins, E., Styrud, J., Ögren, P., Smith, C.: A survey of behavior trees in robotics and AI. Robotics Auton. Syst. **154**, 104096 (2022). https://doi.org/10.1016/J.ROBOT.2022.104096
34. Jung, G., Joshi, K.R., Hiltunen, M.A., Schlichting, R.D., Pu, C.: Generating adaptation policies for multi-tier applications in consolidated server environments. In: 2008 International Conference on Autonomic Computing. pp. 23–32 (2008). https://doi.org/10.1109/ICAC.2008.21
35. Kephart, J.O., Chess, D.M.: The Vision of Autonomic Computing. IEEE Computer **36**(1), 41–50 (January 2003). https://doi.org/10.1109/MC.2003.1160055
36. Klös, V., Göthel, T., Glesner, S.: Comprehensible and dependable self-learning self-adaptive systems. Journal of Systems Architecture **85**, 28–42 (2018)
37. Kohita, R., Wachi, A., Kimura, D., Chaudhury, S., Tatsubori, M., Munawar, A.: Language-based general action template for reinforcement learning agents. In: Zong, C., Xia, F., Li, W., Navigli, R. (eds.) Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021. pp. 2125–2139. Association for Computational Linguistics, Online (Aug 2021). https://doi.org/10.18653/v1/2021.findings-acl.187
38. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Proceedings of the 23rd International Conference on Computer Aided Verification (CAV). vol. LNCS:6806, pp. 585–591 (2011)
39. Li, N., Adepu, S., Kang, E., Garlan, D.: Explanations for human-on-the-loop: A probabilistic model checking approach. In: Proceedings of the IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. pp. 181–187 (2020)
40. Lim, B.Y., Dey, A.K., Avrahami, D.: Why and why not explanations improve the intelligibility of context-aware intelligent systems. In: SIGCHI conference on human factors in computing systems. pp. 2119–2128 (2009). https://doi.org/10.1145/1518701.1519023
41. Little, I., Thiébaux, S.: Probabilistic planning vs replanning. In: Proc. of ICAPS Workshop on IPC: Past, Present and Future (2007)
42. Mayr, M., Chatzilygeroudis, K., Ahmad, F., Nardi, L., Krueger, V.: Learning of parameters in behavior trees for movement skills. In: 2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). pp. 7572–7579 (2021). https://doi.org/10.1109/IROS51168.2021.9636292

43. McCabe, T.J.: A complexity measure. IEEE Transactions on software Engineering (4), 308–320 (1976)
44. McMillan, K.L.: Symbolic Model Checking. Springer US, Boston, MA (1993). https://doi.org/10.1007/978-1-4615-3190-6_3
45. Minato, S.i.: Binary decision diagrams and applications for VLSI CAD. Kluwer Academic Publishers, USA (1996)
46. Mitchell, T.M.: Machine learning. McGraw Hill series in computer science, McGraw-Hill (1997)
47. Ouedraogo, L., Kumar, R., Malik, R., Akesson, K.: Nonblocking and safe control of discrete-event systems modeled as extended finite automata. IEEE Transactions on Automation Science and Engineering **8**(3), 560–569 (2011). https://doi.org/10.1109/TASE.2011.2124457
48. Padalkar, A., Quere, G., Steinmetz, F., Raffin, A., Nieuwenhuisen, M., Silvério, J., Stulp, F.: Guiding reinforcement learning with shared control templates. In: 2023 IEEE International Conference on Robotics and Automation (ICRA). pp. 11531–11537 (2023). https://doi.org/10.1109/ICRA48891.2023.10161058
49. Päßler, J., ter Beek, M.H., Damiani, F., Tapia Tarifa, S.L., Johnsen, E.B.: Formal Modelling and Analysis of a Self-Adaptive Robotic System. In: Herber, P., Wijs, A. (eds.) Proceedings of the 18th International Conference on integrated Formal Methods (iFM 2023), LNCS, vol. 14300, pp. 343–363. Springer (2023). https://doi.org/10.1007/978-3-031-47705-8_18
50. Plambeck, S., Fey, G., Schyga, J., Hinckeldeyn, J., Kreutzfeldt, J.: Explaining Cyber-Physical Systems Using Decision Trees. In: 2022 2nd International Workshop on Computation-Aware Algorithmic Design for Cyber-Physical Systems (CAADCPS). pp. 3–8 (2022). https://doi.org/10.1109/CAADCPS56132.2022.00006
51. Quinlan, J.R.: Induction of decision trees. Mach. Learn. **1**(1), 81–106 (1986)
52. Rezende Silva, G., Päßler, J., Zwanepol, J., Alberts, E., Tapia Tarifa, S.L., Gerostathopoulos, I., Johnsen, E.B., Hernández Corbato, C.: SUAVE: An Exemplar for Self-Adaptive Underwater Vehicles. In: Proceedings of the 18th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2023). pp. 181–187. IEEE (2023). https://doi.org/10.1109/SEAMS59076.2023.00031
53. Rodrigues, A., Caldas, R.D., Rodrigues, G.N., Vogel, T., Pelliccione, P.: A learning approach to enhance assurances for real-time self-adaptive systems. In: Andersson, J., Weyns, D. (eds.) Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems, SEAMS@ICSE 2018, Gothenburg, Sweden, May 28-29, 2018. pp. 206–216. ACM (2018). https://doi.org/10.1145/3194133.3194147
54. Salehie, M., Tahvildari, L.: Self-Adaptive Software: Landscape and Research Challenges. ACM Transactions on Autonomous and Adaptive Systems **4**(2), 14:1–14:42 (2009). https://doi.org/10.1145/1516533.1516538
55. Schwammberger, M., Klös, V.: From specification models to explanation models: An extraction and refinement process for timed automata. Electronic Proceedings in Theoretical Computer Science **371**, 20–37 (2022). https://doi.org/10.4204/eptcs.371.2
56. Shoulson, A., Garcia, F.M., Jones, M., Mead, R., Badler, N.I.: Parameterizing behavior trees. In: Motion in Games: 4th International Conference, MIG 2011, Edinburgh, UK, November 13-15, 2011. Proceedings 4. pp. 144–155. Springer (2011)

57. Sistla, M.A., Chaudhuri, S., Reps, T.: Cflobdds: Context-free-language ordered binary decision diagrams. ACM Trans. Program. Lang. Syst. (mar 2024). https://doi.org/10.1145/3651157, just Accepted
58. Srinivasan, A., Ham, T., Malik, S., Brayton, R.: Algorithms for discrete function manipulation. In: 1990 IEEE International Conference on Computer-Aided Design. Digest of Technical Papers. pp. 92–95 (1990). https://doi.org/10.1109/ICCAD.1990.129849
59. Sun, Y., Yin, X., Huang, F.: Temple: Learning template of transitions for sample efficient multi-task rl. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 35, pp. 9765–9773 (2021)
60. Union, E.: On the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46/ec (general data protection regulation). In: Regulation (EU) 2016/679 of the European Parliament And of the Council. vol. Article 13(2)(f) (2016)
61. Watson, A., Wallace, D., McCabe, T., Associates, M.., of Standards, N.I., (U.S.), T.: Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. No. v. 13 in NIST special publication, U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology (1996)
62. Weyns, D.: An Introduction to Self-Adaptive Systems: A Contemporary Software Engineering Perspective. John Wiley & Sons (2020)
63. Zapreev, I.S., Verdier, C., Mazo, M.: Optimal symbolic controllers determinization for BDD storage. IFAC-PapersOnLine **51**(16), 1–6 (2018). https://doi.org/https://doi.org/10.1016/j.ifacol.2018.08.001, 6th IFAC Conference on Analysis and Design of Hybrid Systems ADHS 2018
64. Zuse, H.: Software Complexity: Measures and Methods. Programming complex systems, W. de Gruyter (1991)