Towards Adaptive Semantic Subscriptions for Stream Reasoning in the Robot Operating System

Daniel de Leng* Fredrik Heintz*

Abstract— **Modern robotic systems often consist of a growing set of information-producing components that need to be appropriately connected for the system to function properly. This is commonly done manually or through relatively simple scripts by specifying explicitly which components to connect. However, this process is cumbersome and error-prone, does not scale well as more components are introduced, and lacks flexibility and robustness at run-time. This paper presents an algorithm for setting up and maintaining implicit subscriptions to information through its semantics rather than its source, which we call semantic subscriptions. The proposed algorithm automatically reconfigures the system when necessary in response to changes at run-time, making the semantic subscriptions adaptive to changing circumstances. To illustrate the effectiveness of adaptive semantic subscriptions, we present a case study with two SoftBank Robotics NAO robots for handling the cases when a component stops working and when new components, in this case a second robot, become available. The solution has been implemented as part of a stream reasoning framework integrated with the Robot Operating System (ROS).**

I. INTRODUCTION

Robots are increasingly forced to share their operational environment with humans who can cause harm to, or be harmed by, those robots. Providing a safe environment for robots and humans alike is important. *Stream reasoning* can help by e.g. performing execution monitoring. This can be used to ensure that a robot's behavior is in accordance with its specifications. Stream reasoning is understood as incremental reasoning with incrementally-available information.

As a concrete application domain, consider the RoboCup Standard Platform League (SPL) in which teams of SoftBank Robotics NAO robots are tasked with playing soccer with and against each other. Different game states have different playing styles associated with them. Recognizing when the game state changes is important in order to respond effectively. For example, if the ball is on the opponent's side and one of our team's robots has control of the ball, we can go on the offensive. However, if the ball leaves the opponent's side of the soccer field for too long, we should perhaps switch to a defensive posture and pull back our forward players. These rules can elegantly be expressed using a combination of temporal formalisms, such as Metric Temporal Logic (MTL) [1], and spatial formalisms, such as the Region Connection Calculus (RCC8) [2]. By checking whether the ball does not leave the opponents' side for more than a certain amount of time, and registering a violation of such a rule, we can respond appropriately. It can however be a challenge to perform this type of reasoning if a robot's configuration is fixed. The larger an autonomous system gets, the more components it is composed of, and the connections between components get confusing and error-prone. In these cases, reasoning about streams and their provenance can help by allowing us to reorganize the stream processing to respond to reasoning needs.

While scenarios such as this one may seem rather contrived, the problems they illustrate become even more relevant when taken to the scale of smart cities or the Internet of Things (IoT). Many present-day stream-based systems subscribe to information by its source rather than by its semantics, making them fragile in cases where the subscribed-to information's quality deteriorates with environmental changes (e.g. day-night cycle, weather) or even the addition, replacement, or removal of information sources.

The main contribution of this paper is a formalization and concrete realization of adaptive semantic subscriptions, which subscribe to information based on its semantics rather than its source and periodically evaluate the quality of the subscriptions. Adaptive semantic subscriptions can be used to robustly generate state streams over which stream reasoning can be performed. A concrete realization of the framework is given by the latest generation of the DyKnow-ROS stream reasoning framework[1], which previously extended the Robot Operating System (ROS) with reconfigurability support for nodelets [3]. The proposed adaptive semantic subscriptions were applied to and tested with Linköping University's RoboCup SPL implementation.

The remainder of this paper is organised as follows. Related work is presented in Section II, followed by a formal model of the stream reasoning framework and semantic subscriptions in particular in Section III. A concrete realization of this formal model is presented in Section IV. Section V presents a real-world robot case study wherein SoftBank Robotics NAO platforms are used to monitor a ball on a RoboCup soccer field while their configurations change during run-time. Finally, the paper concludes in Section VI with a recap and a discussion of potential future work.

II. RELATED WORK

The DyKnow-ROS realization of the stream reasoning framework presented in this paper is a descendent of the original DyKnow stream reasoning framework [4], [5], which was integrated with the Common Object Request Broker Architecture (CORBA) and lacked support for semantic

*Department of Computer and Information Science, Linköping University, Sweden, email: {daniel.de.leng, fredrik.heintz}@liu.se

[1]DyKnow-ROS website: http://www.dyknow.eu

subscriptions. A comprehensive survey on other stream reasoning approaches is given by Cugola and Margara [6], who cover both Data Stream Management Systems (DSMS) and Complex Event Processing (CEP) systems. The survey precedes recent work on analyzing stream processing as done by for example SECRET [7] or more recently for stream reasoning by LARS [8], [9], which complement our formalization of semantic subscriptions.

A related approach was proposed by Bröring et al., who identified challenges to achieving semantically-enabled sensor plug-and-play [10]. They proposed a method for plug-and-play functionality by making use of a Sensor Bus [11] that matches services to sensors. Research towards Semantic Sensor Networks led to the development of the Semantic Sensor Network ontology (SSN) [12], which focuses on well-structured semantic descriptions of sensors. Our work makes use of semantic descriptions of streaming components rather than sensors by using functional descriptions of the inputs and outputs of these components. These functional descriptions are extensions of the OWL-S service ontology [13] applied to a streaming context. Our proposed solution for semantic subscriptions is more advanced than the Sensor Bus approach in that we periodically recombine and reconnect components whereas the Sensor Bus directly connects with information sources.

These reconfiguration capabilities are closely related to configuration planning. Automatic (re)configuration techniques have been studied in detail [14], [15], [16]. The work by Tang and Parker [17] on ASyMTRe is an example of a system geared towards the automatic self-configuration of robot resources in order to execute a certain task. Similar work was performed by Lundh, Karlsson and Saffiotti [18] related to the Ecology of Physically Embedded Intelligent Systems, also called the PEIS-ecology [19]. Given a high-level goal describing as a task, Lundh et al. use a configuration planner to configure a collection of robots towards the execution of the task rather than logic-based stream reasoning. Their solution is however designed for use within the PEIS middleware and does not easily transfer to the ROS middleware. Lundh [20] further points out that their approach uses static cost measures and could benefit from incorporating semantic knowledge. Our approach focuses on a more advanced representation of cost, and makes use of semantic descriptions for components.

III. FORMAL MODEL

We use the term *stream reasoning framework* to mean a generic framework that is used for the refinement of streams for purposes of stream reasoning. The primary task of a stream reasoning framework is thus to make possible and maintain stream reasoning. A high-level overview of the architecture is shown in Fig. 1. It considers three primary modules: a stream reasoning manager (SRM), a stream reasoning engine (SRE), and a computational environment. A client send queries to the SRM (1), which in turn reconfigures the computational environment (2) to produce streams required to answer the query. This is achieved by adding, removing,

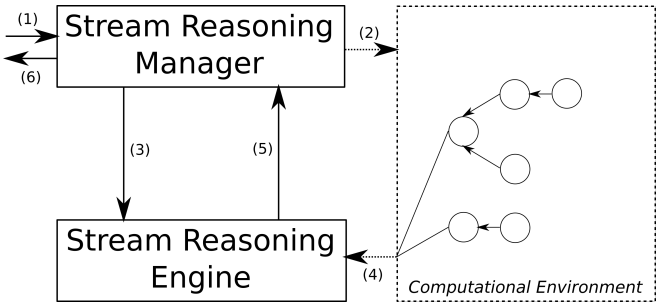

Fig. 1. Architecture of the proposed stream reasoning framework.

and reconnecting the resident components, which are tasked with stream refinement. The query is then forwarded to the SRE (3) together with the semantic subscriptions (4) that were set up by the SRM earlier. Once the SRE generates an answer it is returned to the SRM (5) to handle any side-effects such as unloading components that are no longer needed; and to external entities such as the client (6).

We present a formal model of the SRM and the computational environment, which deal with setting up and maintaining semantic subscriptions. The SRE, which will not be covered in further detail for the sake of brevity, performs *progression* over MTL formulas in accordance with the procedure proposed by Bacchus and Kabanza [21], [22]. Progression incrementally evaluates an MTL formula through syntactic rewritings based on the information received thus far, thereby making it possible to potentially draw a conclusion prior to the receipt of the entire stream. Semantic subscriptions provide streams over which progression can be performed once synchronized into a single stream of states.

A. Computational Environment

We formally represent a stream reasoning framework by a *computation graph*, *transformations* and *targets*. The computation graph represents the computational environment consisting of *computation units* connected by *streams*.

Definition 1 (Stream): A *stream* is an unbounded sequence of time-stamped values $((l_0, v_0, t_0), (l_1, v_1, t_1), \dots)$ where $v_i \in \mathcal{V}$ represent (structured) values, $l_i \in \mathsf{Var}$ represent variable names, and $t_i \in T$ represent time-points.

Streams thus represent information flows over a transportation mechanism, which we refer to as a *channel*. Streams are the product of transformations, which can either refine existing streams into new streams, or act as sources by generating streams without requiring any input streams. In practice, sources often use information external to the computational environment to generate streams, for example through sensor observations. A transformation is considered to be an annotated streaming function that needs a context to perform its operations, whereas a computation unit is the application of such a streaming function to a specific context.

Definition 2 (Transformation): A *transformation* (TF) is an annotated stream-generating function that takes streams as inputs. It is described by a tuple

$$\langle tid, f(x_1, \dots, x_n, \mathcal{S}), [itag_1, \dots, itag_n], otag \rangle ,$$

where $tid \in \mathbb{N}$ represents a unique transformation identifier, $f : \mathcal{V}^n \times \mathcal{S} \hookrightarrow \mathcal{V} \times \mathcal{S}$ represents a partial function from input values and an initial state to an output value and a resulting state, $itag_i \in \mathsf{Tag}$ represent tags for inputs, and $otag \in \mathsf{Tag}$ represents the output tag.

Definition 3 (Computation Unit): A *computation unit* (CU) is a component that is described by a tuple

$$\langle cid, tid, [in_1, in_2, \ldots, in_n], out, \mathcal{S} \rangle ,$$

where $cid \in \mathbb{N}$ represents a unique identifier for CUs, $tid \in \mathbb{N}$ represents the unique identifier of the transformation which this CU is an instance of, $in_i \in \mathbb{N} \cup \{\mathsf{none}\}$ represent incoming channels, $out \in \mathbb{N} \cup \{\mathsf{none}\}$ represents the outgoing channel, and $\mathcal{S} \subseteq \mathsf{Var} \times \mathcal{V}$ represents the state as a relation between variables and values.

Lastly, the computational environment contains *targets*, which describe semantic subscriptions for outside modules such as the SRE. Note that subscriptions also occur *within* the computational environment, but that these are not referred to as targets because they do not reflect the global configuration goals of the computational environment.

Definition 4 (Target): A *target* describes a desired semantic subscription and is denoted by a tuple $\langle qid, tag, chan \rangle$, where $qid \in \mathbb{N}$ is a unique (query) identifier, $tag \in \mathsf{Tag}$ is a description of the desired information, and $chan$ is the channel the described stream is expected on.

Targets thus indirectly represent configuration goals for the computational environment by indirectly referencing desired streams by their semantic descriptions. These streams are generated by transformation, which in turn have input requirements. For a given set of targets, there may be many different computation graphs which satisfy all of the input requirements and similarity relations. However, these graphs likely have different costs associated with them.

By combining these elements, we can formally describe the concept of a computational *environment*.

Definition 5 (Environment): An *environment* is denoted by a tuple $\varepsilon = \langle CU, F, T, \sim \rangle$, where CU denotes a set of computation units called a *computation graph*, F denotes a collection of transformations called a *library*, T denotes a set of targets called a *goal*, and $\sim \subseteq \mathsf{Tag} \times \mathsf{Tag}$ denotes a *similarity relation* between tags. Elements of environment ε have short-hand representations CU_ε, F_ε, T_ε, and $\sim_\varepsilon$ respectively.

B. Dynamics

An environment is a representation of the configuration of the computational environment, which may be subjected to changes over time. These changes are represented by a *change set*.

Definition 6 (Change Set): A *change set* is a tuple

$$\delta = (CU^+, CU^-, F^+, F^-, T^+, T^-)$$

consisting of set additions and set removals denoted by superscript '$+$' and '$-$' respectively.

Whenever an environment changes we call this an *update*. An update is the result of applying a *change* to an environment, yielding a new environment.

Definition 7 (Update): An *update* applying a change set δ to an environment ε is denoted by $\varepsilon' = \varepsilon \otimes \delta$ (alternatively: $\varepsilon \Rightarrow_\delta \varepsilon'$), where $\otimes$ maps environments ε and change set δ to resulting environments ε' such that

$$\begin{aligned} CU_{\varepsilon'} &= (CU_\varepsilon \cup CU_\delta^+) \setminus CU_\delta^- , \\ F_{\varepsilon'} &= (F_\varepsilon \cup F_\delta^+) \setminus F_\delta^- , \\ T_{\varepsilon'} &= (T_\varepsilon \cup T_\delta^+) \setminus T_\delta^- . \end{aligned}$$

Updates thus apply additions before removals and may take a certain amount of time to complete. They also have an immediate cost associated with them based on the individual instantiation costs of transformations, referred to as *labor*, and a latent cost from the *upkeep* requirements of the resulting computation graph. The combination of the one-time labor cost estimation with the run-time upkeep cost estimation gives us a cost estimator for updates.

Definition 8 (Cost estimator): The cost estimator $\widehat{cost}$ combining estimators $\widehat{upkeep}$ and $\widehat{labor}$ is defined as

$$\widehat{cost}(\varepsilon, \delta, \tau, H) = \widehat{labor}(\delta, \tau) + \sum_{t=\tau+1}^{\tau+H} \widehat{upkeep}(\varepsilon \otimes \delta, t)$$

for update $\varepsilon \otimes \delta$ at time-point τ with a horizon of H time-units. The cost estimator thus combines short-term labor with long-term upkeep costs taking into account the horizon.

C. Validity and Optimality

As the result of updates, a computational environment ε may become invalid or suboptimal. This may for example happen due to changing operational costs associated with CUs (upkeep), CUs may crash and require replacing, transformations may become unavailable rendering their CU instances invalid, or new transformations may become available for a lower cost. In order to maintain adaptive semantic subscriptions, the problem is to find a change set such that, when applied to ε, the resulting environment is valid and update is optimal.

Definition 9 (Validity): An environment ε is *valid*, denoted by $\varepsilon \in \mathsf{Valid}$, iff for every CU:

1) there exists an associated TF in F;
2) for every identifier in_i there exists a CU in CU for every $1 \leq i \leq n$, i.e. no subscriptions to none;
3) for every target $\langle qid, tag, chan \rangle$ in ε, there exists a CU with an associated TF such that $tag \sim_\varepsilon otag$; and
4) $itag_i \sim_\varepsilon otag$ holds for every connected pair of CUs.

We can exclude change sets that yield an invalid environment when used in an update. This reduces the number of applicable change sets to just those that yield environments that satisfy all targets. An *optimal* update is one that minimizes the estimated cost of applying a change set and the estimated upkeep over a predetermined horizon.

Definition 10 (Optimality): An update $\varepsilon \otimes \delta^*$ yielding a valid environment is *optimal* relative to a horizon of H time-

units at time-point τ iff $\delta^* \in \Delta^*$, where

$$\Delta^* = \arg\min_{\delta} \ \widehat{cost}(\varepsilon, \delta, \tau, H)$$
$$\text{subject to } \widehat{cost}(\varepsilon, \delta, \tau, H) \leq \texttt{MAX\_COST}$$

for cost estimator $\widehat{cost}$ and upper bound MAX_COST.

Note that there may be many optimal change sets, in which case any can be chosen. Alternatively, if no change set can make the resulting environment valid ($\Delta^* = \varnothing$), there are no optimal change sets. The choice of horizon determines how conservative change sets are; if the horizon is large, upkeep starts to outweigh cost more than in cases where the horizon is kept short. Different estimators can be used, ranging from simplistic constant values to advanced predictive models whose accuracy is used to increase or decrease the length of the horizon. By performing an optimal update at the end of each horizon, the system adaptively establishes and maintains semantic subscriptions described by the (possibly changing) set of targets.

IV. IMPLEMENTATION

The latest version of DyKnow-ROS is concrete realization of a stream reasoning framework integrated with ROS and implements adaptive semantic subscriptions. To implement CUs it makes use of extended ROS nodelets, which are procedures that can be started (loaded) and stopped (unloaded) during run-time. Nodelets are hosted in a nodelet manager which runs the procedures using a thread pool. ROS nodes, on the other hand, are themselves run as processes rather than threads. Since there exists no built-in services for loading or unloading nodes during run-time, nodelets were chosen instead. Nodelets and nodes alike publish on and subscribe to topics, which are named channels over which data flows. Our earlier work [3] extended ROS nodelets with services that allow for the run-time reconfiguration of these publishers and subscribers by changing their target topics. The SRM makes use of these services to perform updates as presented in the formal model.

A. Transformation and Target Specifications

Transformations from the formal model are realized by *transformation specifications*. The SRM keeps track of these specifications in addition to the computation graph and targets. For practicality, we allow for nodelets to have multiple outputs, which formally corresponds to multiple transformations that take the same inputs performed by a single CU.

The specifications contain a reference to a nodelet's implementation, which is required to load that nodelet. Additionally, a configuration and label are added. An example of a transformation specification is shown in Listing 1. The *label* of a transformation corresponds to a tid. The *source* refers to the path used by the ROS nodelet manager to dynamically load a nodelet. The *parameters* are passed to the nodelet after loading, when the SRM sets its configuration. A listing of *ports* is provided, corresponding to input and output indices which are given a programmer-friendly name

Listing 1. Transformation specification example

```
<transformation type="nodelet">
  <label>undistort(cam1)</label>
  <source>package/Undistort</source>
  <params>
    <param name="configPath" type="string">
      /path/to/configuration/cam1/
    </param>
  </params>
  <ports>
    <port type="out">undist</port>
    <port type="in">rawCamera</port>
  </ports>
  <tags>
    <tag port="undist">Undistorted(cam1)</tag>
    <tag port="rawCamera">RawRGB(cam1)</tag>
  </tags>
</transformation>
```

Listing 2. Target specification example

```
<target>
  <label>undistortSub</label>
  <topic>/result</topic>
  <tag>Undistorted(cam1)</tag>
</target>
```

that can be used in the program code. These ports can subsequently be annotated using tags from the Tag set. In the example in Listing 1, first-order predicates are used as tags to describe a transformation which takes RGB image data and produces undistorted images based on a lens model which was provided as part of the configuration.

A *target specification* is similar to a transformation specification. An example target specification is shown in Listing 2. It is given a programmer-friendly name undistortSub, and requires a stream of undistorted images from cam1 to be sent over a ROS topic labelled `/result`. Components external to the computational environment, such as the SRE, can then subscribe to these ROS topics with the assurance that they carry the desired information.

B. Semantic Tagging and Matching

A different approach to tagging is to use ontological concepts from the Semantic Web [23]. We can specify complex concepts in Description Logic (DL) which are expressible in various syntaxes. The semantics of the undist output port from Listing 1 can be expressed as

$$\mathsf{Undistorted} \ \sqcap \ \exists \mathsf{hasSource.cam1}$$

in DL syntax (using punning), i.e. the intersection of individuals of the concept Undistorted with the individuals for which there is a hasSource relation to cam1. Similarly, the semantics of the rawCamera input port can be expressed as

$$\mathsf{RawCamera} \ \sqcap \ \exists \mathsf{hasSource.cam1}$$

in DL syntax. Turtle syntax or Manchester syntax could be used instead of DL syntax for the XML specifications.

If we want to check whether a transformation with an output port tag C can be subscribed to for the rawCamera input port, we have to establish whether

$$C \sim (\mathsf{RawCamera} \ \sqcap \ \exists \mathsf{hasSource.cam1})$$

holds. For semantic matching with DL concepts, we therefore use $C_{out} \sim C_{in} \equiv_{def} C_{out} \sqsubseteq C_{in}$ where C_{out} denotes the output port tag and C_{in} denotes the input port tag.

We use the same subsumption relation to determine validity for all targets. A valid target is thus any (complex) DL concept. For example, the target containing the tag

$$\mathsf{Undistorted} \sqcap \exists \mathsf{hasSource.Camera}$$

would match the undistort(cam1) transformation's undist port if cam1 $\in$ Camera (i.e. Camera(cam1) holds).

C. Configuration Life-Cycle

The proposed semantic subscriptions are adaptive in the sense that they are periodically evaluated and updated to repair or improve the underlying computation graph. This recurring process is referred to as the *configuration life-cycle*. This configuration life-cycle is composed of a number of phases which are repeated every cycle, which starts with a *review interval* followed by a *stable interval*. During the review interval, the SRM searches for a change set δ^* such that its application to the current environment constitutes an optimal update. Whether an update is optimal is determined by a combination of instantiation costs and upkeep relative to a horizon. DyKnow-ROS uses a fixed horizon length and uses CPU time (combined user and system time) for instantiation cost and upkeep.

This search is followed up with the execution of the found optimal update. Once the update has been performed, the stable interval begins. During the stable interval, the SRM waits for a *perturbation* to occur while the computational environment is left unchanged. A perturbation is said to occur whenever a CU unexpectedly gets unloaded, subscriptions between CUs are unexpectedly removed, targets are added, or the horizon is reached. The first three types of perturbations are premature (they occur potentially well before the horizon is reached) but potentially result in the environment to be invalid, which needs to be mitigated quickly with the start of a new cycle. The last perturbation is the normal way for a cycle to end and a new one to start, as it is used in determining the most cost-efficient change set. Premature perturbations may therefore be expensive whereas horizon-based perturbations are anticipated.

At the end of every cycle, the estimators for labor and upkeep are updated by the SRM based on the weighted recent history of observed labor and observed upkeep. The estimators for upkeep are averaged over the various observed CUs clustered by tid. The initial estimations for previously unobserved labor and upkeep however require bootstrapping with an initial guess. One can encourage the SRM to explore or exploit by assigning very low or very high initial guesses respectively. These labor and upkeep estimators will then be updated with more accurate values when the SRM tries to use them. When new transformations become available during run-time, these too are then either preferred or avoided depending on the bootstrapping strategy used.

Algorithm 1: Exploration procedure

```
1  function EXPLORE (Environment ε) :
2  root ← Node(nil, nil)
3  i ← 1
4  queue ← Queue()
5  foreach target ∈ T_ε do
6  |   targetNode ← Node(root, target)
7  |   addOption(root, targetNode, i)
8  |   i ← i + 1
9  |   enqueue(queue, EXPAND(targetNode, ε))
10 end
11 while |queue| > 0 ∧ ¬TIMEOUT do
12 |   node ← dequeue(queue)
13 |   enqueue(queue, EXPAND(node, ε))
14 end
15 return COMPILE (root, ε)
```

D. Update Procedure

Whenever a perturbation is detected, either due to the horizon being reached or otherwise, the SRM performs its update procedure. The task of the update procedure is to find a change set δ^* for the current environment ε with which an optimal update $\varepsilon \otimes \delta^*$ can be performed. This change set is approximated by exploring the search space for change sets such that the resulting environment $\varepsilon \otimes \delta^*$ is in the Valid relation.

The search algorithm used is based on breadth-first (backwards) search with branch-and-bound, which is followed up by some post-processing to eliminate any duplicate subtrees. The nodes in the tree correspond to either transformations to be instantiated or existing CUs. The edges then correspond to publisher and subscriber pairs.

When a perturbation is detected, the first step is to check whether the set of transformations has changed. If they have, we generate a new *validity graph*. A validity graph is a directed graph where nodes correspond to transformations and labelled edges correspond to whether the output tag and input tag are sufficiently similar ($\sim$). For example, if an output tag for transformation t_1's output index i is C_1 and an input tag for transformation t_2's input index j is C_2, then there exists an edge $((t_1, i), (t_2, j))$ iff $C_1 \sqsubseteq C_2$. This graph can be compact represented as a *validity matrix* where rows and columns correspond to inputs and outputs, allowing for constant time look-up. Rows and columns should be added or removed whenever transformations are added or removed in order to stay up-to-date.

The procedure starts as shown in Algorithm 1 by generating a root node for which the arity is set to $|T|$, i.e. equal to the number of targets. For every target, any transformations that satisfy that target are added as candidates for their associated query and added to the expansion queue. We now have a tree consisting of a root where for every target its satisfying transformations are children associated with that input. The second step is to expand the tree breadth-first. This is done for every element in the expansion queue in sequence, where for every such transformation we consider each of its inputs. The algorithm for expansion is shown in

Algorithm 2: Node expansion

```
1  function EXPAND (Node node, TF[] transforms, Queue
     queue) :
2  arity ← getArity(getType(node))
3  fwdCost ← getFwdCost(getParent(node)) +
     getCost(getType(node))
4  bestCostChain ← getBestCostChain(node)
5  for i ← 1 to arity do
6  |  bestCost ← getBestCost(node, i)
7  |  candidates ← getValid(getType(node), i)
8  |  foreach c ∈ candidates do
9  |  |  cost ← getCost(c)
10 |  |  if cost + fwdCost < bestCostChain then
11 |  |  |  option ← Node(node, c)
12 |  |  |  optionArity ← getArity(c)
13 |  |  |  addOption(node, option, i)
14 |  |  |  if optionArity > 0 then
15 |  |  |  |  enqueue(queue, option)
16 |  |  |  else
17 |  |  |  |  setBestCost(option, getCost(c))
18 |  |  |  |  Propagate new best cost up the tree
19 |  |  |  end
20 |  |  end
21 |  end
22 end
```

Algorithm 2. For every input, the validity matrix is queried for the row associated with the transformation-input pair, yielding a set of transformation-output pairs that are valid for feeding into the input that is under consideration. These candidate transformations are used for two purposes;

1) To check whether there exist CUs in CU_ε for which the type matches the candidate transformation and which we can re-use;
2) To check whether we can instantiate a new computation unit of the candidate transformation.

The cost for re-using a computation unit is assumed to be free; no work needs to be done to connect to a stream it is already producing, although it still generates upkeep. Candidates are rejected if they are known to exceed the currently-known best solution in terms of estimated cost and estimated upkeep relative to the horizon. If the candidate leads to a new valid subtree, its cost is propagated up the tree to keep track of new solutions. The algorithm recursively calls `EXPAND` for both TFs and CUs. CUs, like TFs, are also expanded to ensure they form a valid subtree without gaps caused by CUs having been unloaded.

The algorithm terminates whenever the entire space has been searched or when a time-out is reached. Time-out limits are provided every time the update procedure is run, and may be adjusted between cycles. For the resulting tree there may exist common subtrees that could have been shared. This is a problem as this unnecessarily instantiates duplicate transformations, leading to a greater cost and a greater upkeep. A post-processing step therefore recursively steps through the tree breadth-first considering only the best-found candidate for every input, and indexes these nodes. If a subtree is encountered for which the root has already been indexed, the subtree is replaced with a pointer to the root of the earlier-found duplicate subtree. This pointer tells the algorithm to subscribe to whatever stream is being produced by that subtree, avoiding duplicate subtrees.

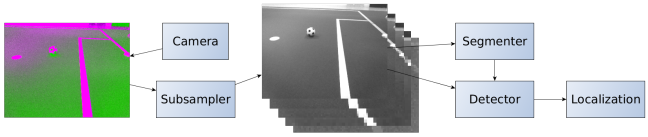

Fig. 2. Piff and Puff's transformation pipeline conceptually showing the transformations from camera images to ball positions.

In the final step of the algorithm, we end up with a trimmed-down tree that the algorithm now steps through depth-first, limiting itself to the best-found candidates as previously. For every node, it determines unique topic names to subscribe to, and recursively informs its children of these names so they can publish to those topics. In stepping through the tree, the algorithm incrementally builds a change set of transformations to instantiate and topics for CUs to subscribe or produce to. Since the existing CUs are taken into account, the algorithm keeps a list of CUs that are going to be re-used, protecting them from being removed. Any CUs that end up not being protected are subsequently added to the change set for removal, reducing the global upkeep. This change set is then returned as the result of the algorithm, and can be applied to the environment to approximate an optimal update.

Since the exploration procedure applies an exhaustive search of valid configurations, the time and memory requirements are closely tied to the average branching factor of the search tree and the granularity of the transformations. The branching factor is in turn determined by the similarity relation and semantic annotations used. In other words, if every transformation can be connected to every other transformation, the branching factor will be high and the time to complete a search will go up. Likewise, the more fine-grained transformations are, the deeper the search will have to go before finding valid change sets. In practice, software components are not so general that they could be connected to *any* other component. Data types alone impose constraints on what can be connected; semantic constraints further limit the number of candidates. The time-out mechanism further makes it possible to conclude the exploration procedure early.

V. CASE STUDY

Our case study focuses on two NAO robots, called *Piff* and *Puff* (Swedish for *Chip 'n Dale*). Both Piff and Puff are capable of running a processing pipeline that takes in sensor information and produces ball coordinates relative to the soccer field. This series of transformations is shown conceptually in Fig. 2. For the case study, we are interested in situations where semantic subscriptions can provide added value to Piff in performing its task of tracking the ball. We consider two cases; 1) Piff is tracking the ball but something goes wrong; and 2) Piff is tracking the ball and Puff offers to help for a while. Piff and Puff are assumed to be part of

TABLE I

TRANSFORMATION ANNOTATIONS FOR GENERATING BALL POSITIONS

(1)	Label:	bottom_cam($\${NAO}$)
	In:	$\varnothing$
	Out:	YUVImage $\sqcap$ $\exists$hasSource.$\${NAO}$
(2)	Label:	subsampler
	In:	YUVImage $\sqcap$ $\exists$hasSource.$\${NAO}$
	Out:	Y640X480Image $\sqcap$ $\exists$hasSource.$\${NAO}$, ...,
		U640X480Image $\sqcap$ $\exists$hasSource.$\${NAO}$, ...,
		V640X480Image $\sqcap$ $\exists$hasSource.$\${NAO}$, ...,
(3)	Label:	segmenter
	In:	Y80X60Image $\sqcap$ $\exists$hasSource.$\${NAO}$,
		V80X60Image $\sqcap$ $\exists$hasSource.$\${NAO}$
	Out:	ConvHull $\sqcap$ $\exists$hasSource.$\${NAO}$
		$\sqcap\exists$hasTarget.field
(4)	Label:	ball_detector
	In:	ConvHull $\sqcap$ $\exists$hasSource.$\${NAO}$
		$\sqcap\exists$hasTarget.field,
		Y640X480Image $\sqcap$ $\exists$hasSource.$\${NAO}$,
		Y320X240Image $\sqcap$ $\exists$hasSource.$\${NAO}$
	Out:	PixelPos $\sqcap$ $\exists$hasSource.$\${NAO}$
		$\sqcap\exists$hasTarget.Ball
(5)	Label:	ball_localization
	In:	PixelPos $\sqcap$ $\exists$hasSource.$\${NAO}$
		$\sqcap\exists$hasTarget.Ball,
		Pose $\sqcap$ $\exists$hasSource.$\${NAO}$
	Out:	Position $\sqcap$ $\exists$hasSource.Ball

the same computational environment; a multi-agent system approach is beyond the scope of this paper.

A. Initial Set-Up

Initially, Piff is the only NAO that will try to continuously report the position of the ball. A target is provided by a user with a tag Position $\sqcap$ $\exists$hasSource.Ball in DL, and a topic `/result`. At this time, no transformations are known to the SRM, so the update amounts to adding the new target $\langle qid_0, \text{Position} \sqcap \exists\text{hasSource.Ball}, \texttt{/result}\rangle$ to T_ε. The resulting environment is already optimal since there are no known transformations satisfying the target.

Later on, Piff gets registered to the SRM by reporting its transformations as being available for use. Due to space limitations, we cannot show the full XML transformation specifications. However, the ball detection pipeline's tags are shown in Table I, where $\${NAO}$ refers to a collection $\text{piff}, \text{puff} \in \text{NAO}$, i.e. Piff and Puff are instances of the NAO concept referenced by the template variable $\${NAO}$.

The bottom_cam TF provides a YUV image stream, which can be subscribed to by the subsampler TF. This transformation down-samples the resolution of the three channels into 640x480, 320x240, 160x120, 80x60, and 40x30. The segmenter TF instances may subscribe to low-resolution Y and V channels to determine the convex hull of the green field, ignoring the space in the image which captures things outside of the field. This convex hull is combined with the Y channel by the ball_detector TF to produce pixel coordinates of balls, which is then combined with pose information by the ball_localization TF to produce ball position data, which matches the query. Since the validity matrix is updated when transformations are added or removed, the result is a 11×19 validity matrix for the 11 inputs and 19 outputs. Based on this new matrix, the ball_localization TF now satisfies the target.

As the result of the perturbation, the planner searches for a solution and finds one as shown conceptually in Fig. 2. The change set is the instantiation of all transformations, and the connection of the resulting CUs in accordance with their annotations. Piff now produces a ball position stream on the `/result` topic, to which for example the SRE can subscribe as an adaptive semantic subscription.

B. Recovery from Failures

Unfortunately, something goes wrong and the image segmenter is unloaded, leaving a hole in the computation graph and interrupting the flow of position information. This perturbation is detected as $CU^- = \{\text{segmenter\_1}\}$. The subsampler is still producing a stream of low-resolution images, but the segmenter no longer exists to do anything with them. The environment is now $\langle CU, F, T, \sqsubseteq\rangle$, where $T = \{\langle qid_0, \text{Position} \sqcap \exists\text{hasSource.Ball}, \texttt{/result}\rangle\}$, and the computation graph CU is as in Fig. 2 but without a segmenter. This perturbation results in the update procedure generating a change set by re-using the part of CU that still exists, but instantiating a new computation unit segmenter_2 and reconfiguring it to subscribe to the streams that were already being produced by the subsampler. This is important because if the publishers of the subsampler were to be reconfigured, the detector would no longer receive anything unless it too is reconfigured. The detector's subscription to the defunct segmenter is then replaced by one to the new segmenter, and the position information flow is restored.

C. Combining Resources

Some time later, Puff joins Piff on the field and registers its own transformations in accordance with the transformation table. Given the possibility to generate a second pipeline for ball positions, the SRM nevertheless does not use the second pipeline as-is. The reason for this is that the cost for re-using Piff's part of the computation graph is assumed to be free, whereas a lot of effort would have to be spent in order to instantiate Puff's pipeline to switch away from Piff's stream. An exception to this occurs when we introduce a transformation fuse_ball_pos, which specifically attempts to fuse two streams containing ball position information into a stream with improved ball position information. By itself, this new TF will not be used because it adds to the upkeep of the computation graph. This is currently a shortcoming of the proposed approach, because we limit ourselves to the cost of updates without taking into consideration the potential for an increase of quality of the information semantically-subscribed to. This problems highlights the difficulty in balancing the cost of updates with the quality of a stream resulting from such an update. Taking into account quality in addition to the cost measurements focused on in this paper is left for future work. A temporary workaround is to explicitly describe fusion transformations. In this example, we could replace the target with one like FusedPosition $\sqcap$ $\exists$hasSource.Ball in DL, and describe the

fusion TF accordingly. Alternatively, the SRM can be forced to use fusion TFs if they are available. In that case, the planner will find three solutions:

1) Keep using Piff's ball position stream;
2) Instantiate and switch to Puff's ball position stream;
3) Instantiate Puff's ball position stream and fuse with Piff's existing ball position stream;

but it is forced to pick the fusion approach despite being more expensive. The update procedure then instantiates a second pipeline as shown in Fig. 2 and connects the resulting position streams as inputs to a domain-specific ball fusion stream, for which the output is produced to `/result`.

VI. CONCLUSIONS

ROS is a powerful middleware for robotic application that provides built-in capabilities to subscribe to and produce data incrementally. However, these subscriptions are static subscriptions based on a topic name rather than a description of the desired information. Furthermore, static subscriptions are not robust to changing conditions during run-time, such as the introduction of additional components or the removal of existing components.

This paper presents an algorithm for semantic subscriptions, which subscribe to information by its semantics rather than its source. Semantic subscriptions are described by targets, and optimal updates seek to apply a change set to an environment such that the resulting environment satisfies the targets. The combined cost of the change set with the upkeep of the resulting environment relative to a horizon is approximately minimized. DyKnow-ROS is a concrete instantiation of a stream reasoning framework that supports semantic subscriptions in the ROS. It makes use of Semantic Web concepts to semantically annotate transformations with tags for inputs and outputs, and applies semantic matching to determine whether subscriptions are valid. ROS nodelets extended with reconfiguration services are used as transformation instances, which can be loaded, unloaded, and the subscriptions of which can be reconfigured during run-time. A concrete update procedure periodically updates the computational environment to cope with changes, using historic CPU time as an estimator for transformation instantiation cost and CU upkeep. DyKnow-ROS is shown to work in a real-world scenario involving two NAO robots tracking a ball, where the computational environment changes due to robots joining or leaving.

Future work can include the consideration of different application areas, such as smart cities and the IoT, as well as various improvements to the proposed model and associated life cycle. For example, more expressive tagging languages can be developed, and cost measures can be extended by taking into consideration the changing quality of streams. The proposed solution is a step forward towards making robotic systems more robust and capable of adapting to changes in their computational environment during run-time.

ACKNOWLEDGMENT

This work is partially supported by grants from the National Graduate School in Computer Science, Sweden (CUGS), the Swedish Aeronautics Research Council (NFFP6), the Swedish Foundation for Strategic Research (SSF) project CUAS, the Swedish Research Council (VR) Linnaeus Center CADICS, and the ELLIIT Excellence Center at Linköping-Lund for Information Technology. Special thanks to the Linköping University RoboCup SPL team for letting us use their ball detection pipeline.

REFERENCES

[1] R. Koymans, "Specifying real-time properties with metric temporal logic," *Real-Time Systems*, vol. 2, no. 4, pp. 255–299, 1990.
[2] D. Randell, Z. Cui, and A. Cohn, "A spatial logic based on regions and connection," in *Proc. KR*, 1992, pp. 165–176.
[3] D. de Leng and F. Heintz, "Dyknow: A dynamically reconfigurable stream reasoning framework as an extension to the robot operating system," in *Proc. SIMPAR*, 2016, pp. 957–963.
[4] F. Heintz and P. Doherty, "DyKnow: An approach to middleware for knowledge processing," *Journal of Intelligent and Fuzzy Systems*, vol. 15, no. 1, 2004.
[5] F. Heintz, "DyKnow : A stream-based knowledge processing middleware framework," Ph.D. dissertation, Linköping University, 2009.
[6] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," *ACM Computing Surveys (CSUR)*, vol. 44, no. 3, p. 15, 2012.
[7] I. Botan, R. Derakhshan, N. Dindar, L. Haas, R. J. Miller, and N. Tatbul, "SECRET: a model for analysis of the execution semantics of stream processing systems," *Proc. VLDB Endowment*, vol. 3, no. 1-2, pp. 232–243, 2010.
[8] H. Beck, M. Dao-Tran, T. Eiter, and M. Fink, "Towards a logic-based framework for analyzing stream reasoning," in *Proc. OrdRing*, 2014.
[9] ——, "LARS: A logic-based framework for analyzing reasoning over streams," in *Proc. AAAI*, 2015.
[10] A. Bröring, K. Janowicz, C. Stasch, and W. Kuhn, "Semantic challenges for sensor plug and play," in *Proc. W2GIS*, vol. 5886, no. 1, 2009, pp. 72–86.
[11] A. Bröring, P. Maué, K. Janowicz, D. Nüst, and C. Malewski, "Semantically-enabled sensor plug & play for the sensor web," *Sensors*, vol. 11, no. 8, pp. 7568–7605, 2011.
[12] M. Compton *et al.*, "The SSN ontology of the W3C semantic sensor network incubator group," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 17, pp. 25–32, 2012.
[13] D. Martin *et al.*, "OWL-S: Semantic markup for web services," *W3C member submission*, 2004.
[14] J. Rao and X. Su, "A survey of automated web service composition methods," in *Proc. SWSWPC*, vol. 3387, no. 1, 2005, pp. 43–54.
[15] S. Dustdar and W. Schreiner, "A survey on web services composition," *International Journal of Web and Grid Services*, vol. 1, no. 1, pp. 1–30, 2005.
[16] E. Pejman, Y. Rastegari, P. M. Esfahani, and A. Salajegheh, "Web service composition methods: A survey," in *Proc. IMECS*, vol. 1, no. 1, 2012, pp. 560–564.
[17] F. Tang and L. Parker, "ASyMTRe: Automated synthesis of multi-robot task solutions through software reconfiguration," in *Robotics and Automation*. IEEE, 2005, pp. 1501–1508.
[18] R. Lundh, L. Karlsson, and A. Saffiotti, "Autonomous functional configuration of a network robot system," *Robotics and Autonomous Systems*, vol. 56, no. 10, pp. 819–830, 2008.
[19] A. Saffiotti, M. Broxvall, M. Gritti, K. LeBlanc, R. Lundh, J. Rashid, B. Seo, and Y.-J. Cho, "The PEIS-ecology project: vision and results," in *Proc. IROS*. IEEE, 2008.
[20] R. Lundh, "Robots that help each other: Self-configuration of distributed robot systems," Ph.D. dissertation, Örebro University, 2009.
[21] F. Bacchus and F. Kabanza, "Planning for temporally extended goals," in *Proc. AAAI*, 1996, pp. 1215–1222.
[22] ——, "Planning for temporally extended goals," *Annals of Mathematics and Artificial Intelligence*, vol. 22, no. 1-2, pp. 5–27, 1998.
[23] T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web," *Scientific American*, vol. 284, no. 5, pp. 34–43, May 2001.