A graphical operations interface for modular surface systems

The MIT Faculty has made this article openly available. Please share how this access benefits you. Your story matters.
Abstract—This paper presents the design and implementation of algorithms for a new graphical operations interface system specifically adapted to operating modular reconfigurable articulated surface systems. Geometric models of heterogeneous robot modules may be connected and disconnected in this interface via drag-and-drop interaction. The resulting assemblies may further be kinematically operated through on-screen direct manipulation. The system maintains a reduced coordinate kinematic model for stability, accuracy, and performance. Key algorithms are presented to evolve this model as the user changes the assembled module topology. Though the presented algorithms are generic, application examples are given primarily for a simulation of NASA/JPL's reconfigurable TriATHLETE system for Lunar exploration. A second application example with a modular robot from the research literature is also included as a demonstration.

TABLE OF CONTENTS
1 INTRODUCTION ................................... 1
2 RELATED WORK ................................ 2
3 USE CASES: TRIATHLETE AND CKbot ...... 2
4 REDUCED COORDINATE KINEMATIC MODEL .... 7
5 ALGORITHMS FOR TOPOLOGICAL EVOLUTION... 9
6 CONCLUSIONS AND FUTURE WORK ............. 10
REFERENCES ...................................... 11
ACKNOWLEDGEMENTS .............................. 11
BIOGRAPHY ....................................... 12

1. INTRODUCTION
Reconfigurable modular hardware, such as the TriATHLETE system currently under development at JPL [1] for Lunar exploration (Figure 1), promises to greatly extend the capability of future surface exploration missions for a relatively small additional cost [2]. Whereas existing missions based on monolithic hardware can only perform a limited set of pre-defined operations, modular hardware can potentially be reconnected and recombined to serve a range of functions. In a Lunar exploration context, such functions could include: lifting and carrying specific payloads; environment manipulation (digging, pushing, drilling); several locomotion modalities; and various crew habitat configurations. Modular hardware also promises to enable graded reduction in mission capabilities in the event of hardware faults—or even total repair when surplus modules are available—and the multi-purpose nature of individual modules can potentially be leveraged for technology reuse across missions.

The full realization of these capabilities will be contingent not just on the development of the hardware itself, but also on the availability of corresponding operations software systems. These must provide algorithms that enable operators to rapidly specify, visualize, simulate, and control (1) particular assemblies of modules, (2) disconnect/reconnect actions that change assembly topology, and (3) feasible motions of any assembly. In the modular robotics research community, this level of on-line operational flexibility has been likened to the need to support “astronauts programming robots” [3]. The more tedious and time consuming aspects of “programming” in traditional environments will clearly need to be addressed in this context, presenting astronauts (or other mission operators) with a rapid and intuitive graphical interface.

Though there is some research literature that at least mentions operator interface software for modular robots [4–12], until now most such interfaces appear to be tightly connected with the specific modular hardware for which they were developed. Often their existence is barely mentioned, with the primary focus of publications typically being the hardware and autonomy algorithms. Little appears to have been published specifically on the detailed design of operator-interface software for modular robots, including general-purpose algorithms that can handle a wide variety of kinematic modules, topological assemblies of modules that can include closed kinematics chains, and click-and-drag direct manipulation with automatic formulation of the general inverse kinematics problem.

This paper aims to address this gap by presenting the design and implementation of key algorithms and data structures for a new graphical operations interface specifically for modular surface systems. These algorithms—and the full system implementation—are generic and applicable across a variety of modular robots. The primary robot we consider is TriATHLE-
LETE, but an extended example is also shown for “CKbot”, a modular robot from the current research literature [13]. Section 3 presents these applications in detail.

In this paper we focus in particular on topological algorithms for maintaining a reduced coordinate kinematic model as modules are connected and disconnected. This contrasts a current trend towards Cartesian coordinate models, which are popular e.g. in game-oriented physics engines [14]. These two types of model are explained in detail in Section 4.

One issue with the Cartesian approach is that it can lead to erroneous misalignments in the on-screen simulation of rigid inter-module connections; using reduced coordinates avoids this by construction in most cases. Reduced coordinate models can also be more computationally efficient as the system typically needs to maintain fewer constraints than a corresponding Cartesian model. Section 4 also presents our reduced coordinate datastructure, and the topological algorithms then follow in Section 5.

We have already presented our topology-independent forward/inverse kinematics computation and click-and-drag direct manipulation algorithms in an earlier publication [15], and do not focus on these aspects of the system here. Full details on both the topological and interactive direct manipulation algorithms are also given in [16].

In [15] we also demonstrated the feasibility of our approach on the original ATHLETE hardware. We have yet to do physical experiments with the new modular TriATHLETE, which is little more than a few months old at the time of this writing. We of course plan to do physical experiments. However, in practice, part of the actual use of this kind of interface software will always be in manipulating simulated models on-screen. Specific operation sequences are engaged on the hardware only after validation in simulation.

2. RELATED WORK

Reports of interactive interface software for specific modular robots have appeared sporadically in the research literature for over two decades [4]. In most cases, the existence of such software is only briefly mentioned, with the primary focus on reporting capabilities of the hardware (or autonomy algorithms) [6, 8–10, 12]. Each interface implementation is typically not portable to other modular robots. The lack of reporting on the software itself may be an indication that it is considered an unglamorous topic (or, said differently, that user interface concerns are not typically the primary interest of researchers in modular robotics). But even though hardware and algorithms for autonomy (autonomous reconfiguration, locomotion, etc.) receive a high profile, in practice there is a real need for interface software that allows operators to efficiently configure modules and to operate the detailed motion of the resulting assemblies. This software is not trivial, given that modular systems can form any kinematic topology, and they may even change topology on-line.

One group that has considered the modular robot interface software problem specifically is Chen et al [5, 7, 17, 18]. Though they present mathematical details for one formulation of arbitrary-topology kinematics computation, they do not appear to present any equivalent to the topological algorithms we cover in Section 5. Moreover, their most recent work [7] appears to be non-graphical, but rather based on a joystick.

Operating a set of kinematically connected modules can also be considered a specific instance of the more general problem of operating a group (or swarm) of robots. Some work has been done in this context, typically focused on locomotion and mobility [19–21]. In particular, Bordignon, Styb, et al have presented a position paper that develops this concept for the case of kinematically connected modules in their ATRON system [11].

3. USE CASES: TRIATHLETE AND CKBOT

TriATHLETE [1] is a recent evolution of the ATHLETE system [22] developed at JPL. Whereas ATHLETE is a single six-limbed robot, TriATHLETE is essentially half of one ATHLETE, with only three limbs. Each TriATHLETE unit is independently functional, and can locomote in a rolling mode. A primary new capability is that multiple TriATHLETE units can approach and dock to a payload from different sides. One typical payload is a rectangular pallet, scaled so that two TriATHLETE units docked to opposite sides reconstructs an analogue to the original 6-limbed ATHLETE (Figure 1).

Figure 1. Rendering from our interface system of two TriATHLETE modules [1] attached to a central rectangular “pallet”. Each TriATHLETE has three limbs, and each limb has 7 revolute kinematic DoF plus a terminal wheel.
**Topological Assembly Operations**

A canonical use-case for our software is thus assembling two TRIATHLETE modules onto a pallet. Figure 2 shows the evolution of the robot model from an initial state with two distinct modules—one TRIATHLETE and one pallet—into a connected configuration. The process is equivalent to adding the second TRIATHLETE module, resulting in the full three-module assembly shown at the top of the figure.

Zooming in to consider the inter-module attachment process in more detail, there are three key transitions. First, the user specifies that there is a rigid connection between the two modules. We model such a connection as a special type of kinematic joint with 0 DoF (no mobility)—a fixed joint, rendered as a red connection in Figure 2 B. Second, the system automatically applies a least-squares numeric solver to move the modules together (Figure 2 C). Third, the connection constraint is effectively converted into an implicit form (Figure 2 E) for all subsequent operations. The semantics of this conversion, which recovers a fully reduced coordinate tree-structured model, is detailed below in Section 5.

It is possible, with significant and tedious effort, to use the modeling features of existing general-purpose CAD, simulation, and animation software to reproduce a similar assembly sequence and the resulting geometric/kinematic model. But due to their generality, these tools can be very time-consuming. In our interface, the full three-module assembly in Figure 2 can be erected in under 10 minutes. Modeling the modules themselves can take longer—it took about half a day to convert an existing 3D model of the 24 DoF TRIATHLETE for use in our system. But such module modeling only has to be done once for each different modular system (another example is given below).

**Kinematic Operations**

At any stage of assembly, the operator may interact with the model to specify kinematic motions. This can be as straightforward and intuitive as clicking on any part of the model and dragging with the mouse, though there are also more quantitative ways to specify kinematic motions in our system. The system will enforce all constraints implied by the robot joints, and additional constraints on coordinated motion can also be defined using virtual joints and links, as we reported previously in [15]. Any subset of the model may also be locked at any time to limit the motion. Figure 3 shows three basic kinematic motions that an operator might specify for the TRIATHLETE-pallet-TRIATHLETE assembly.

In our prior work with the original ATHLETE we demonstrated that motions developed through this kind of kinematic interaction can be directly executed on hardware. We also analyzed and published the least-squares iterative numeric solving approach we use to compute these motions and to maintain kinematic constraints. Full details are given in [16]; we do not focus further on these topics here.

**Demonstrations of Generality**

Our system is of course not limited to just the above canonical assembly. We now demonstrate the generality of our approach in two ways: first, we show the ease with which a novel construction can be formed in the TRIATHLETE modular system (Figure 5); second, we go further and demonstrate an application to a completely different modular robot, CK-bot [13] (Figure 4).

The example in Figure 5 begins where Figure 2 left off: initially there is one TRIATHLETE-pallet-TRIATHLETE assembly. A second pallet is introduced and connected to the side of the first (Figure 5 A-B), and then a third TRIATHLETE is connected (Figure 5 C-D). The result is a novel configuration which could potentially be used for long payloads. Since the TRIATHLETE and pallet modules were already modeled, it took only an additional 10 minutes to create this novel configuration. Configuration-independent click-and-drag kinematic interaction is again easily applied to operate similar lift, slide, and tilt motions as for the canonical three-module assembly (Figure 5 E-G).

The only specialization in our system specific to TRIATHLETE is the set of initially available modules. This can be easily changed, so that a variety of modular kinematic robots can be modeled and potentially operated. Each module can be any open or closed-chain kinematic structure with joints chosen from the 12 available types in Figure 7. Link geometries can be loaded from a variety of standard 3D model formats including VRML97 and COLLADA.

Figure 4 shows an example with the revolute-jointed “CK-bot” L7 and Ubar modules, which are currently known in the research community (e.g., they were used in a hands-on workshop at ICRA 2008 [23]). Compared to TRIATHLETE, these modules are fairly simple, and took only about 2 hours to model from scratch. The 12-module quadruped walker configuration shown in Figure 4 then took only about 10 minutes to assemble and operate in simulation.

---

3 It is possible to enforce realistic kinostatic constraints on the module motion during this assembly, so that the resulting movements could actually be executed on some kinds of hardware, but we do not focus on that here. In practice, other than for self-reconfigurable robots, the assembly of hardware modules involves physical human intervention anyway, e.g. to bolt the modules together.
Figure 2. Parts A-E of this figure show the model evolution as one TriATHLETE module is attached to a rectangular pallet module. Part F shows the final result after attaching two TriATHLETEs to a central pallet. All images are snapshots from our interactive interface software. In the initial state in A, the first TriATHLETE and the pallet are separate kinematic trees (the pallet is a degenerate tree with only a single link) rooted to a common world frame. In the transition from A to B, a new chain-closing 0-DoF “fixed” joint is added at the attachment point between the TriATHLETE and the pallet. The initial gap between them is then reduced automatically by least-squares solving (C), resulting in a structure with a closed kinematic chain (D). The MAKETREE algorithm is then run on the closure joint, resulting in a single tree structure (E) (note the single green “root” joint in E vs the two in D). A similar process is repeated to add the second TriATHLETE module (F).
Figure 3. Any assembly of kinematic modules may be directly operated in our system by click-and-drag direct manipulation. Here the canonical configuration of two TriATHLETE modules and one pallet is operated in lifting (A), sliding (B), and tilting (C) motions. The operator simply specifies that the wheel forks must remain locked in space and then drags the pallet to specify motion. In general, any combination of links and joints can be locked or dragged simultaneously, and quantified motion (“tilt down 30 degrees”) can also be defined using virtual joints and numeric trajectory input [15].

Figure 4. This figure shows the generality of our interface system beyond TriATHLETE. Using the same algorithms as before, a different set of modules is loaded—in this case representing the CKbot system from the modular robotics literature [13]. In part A a graphical model of a 12-module quadruped walker is shown in our interactive interface. The spine is composed of four CKbot “U-bar” modules, and the legs are pairs of “L7” modules. Each module has one revolute DoF, and overall the structure is a kinematic tree rooted at the tail of the walker (B). The root joint is unconstrained, allowing motion of the walker in a global frame. It took only about 2 hours to design geometric and kinematic models for the U-bar and L7 modules from scratch. With these models in place, virtually constructing the walker took about 10 minutes. The modules were assembled graphically without any programming. As before, the resulting model can be operated with click-and-drag direct manipulation, e.g. to design poses for a walking sequence (C, D).
Now that the TriATHLETE and pallet modules have been modeled, new mission scenarios can be rapidly explored, as will likely be necessary in any actual deployment [3]. In this example, a longer structure is created by adding a second pallet and a third TriATHLETE (A-D). It took only a few minutes to create this novel assembly of modules, and the generic click-and-drag direct manipulation interface allows operation of similar lift (E), slide (F), and tilt (G) motions as for the standard single-pallet assembly (Figure 3).
4. Reduced Coordinate Kinematic Model

With the use and function of our system now established, this section and the next go “under the hood” and detail the essential data structures and topological algorithms, respectively, that underlie our generic model for arbitrary-topology and topologically-dynamic modular kinematic systems.

We start with a single module and extend to multiple-module assemblies below. Figure 6 shows the core representation: a kinematic graph where the vertices are links corresponding to rigid bodies and the edges are joints connecting one link to another. A spanning tree is always identified such that there is an identified ground link at its root. If there are any cycles in the graph, then exactly one joint in each cycle is identified as a closure joint not in this spanning tree (all other joints are tree joints). As detailed further below, our model is “reduced coordinate” by virtue of this spanning tree.

![Figure 6. A kinematic graph with 11 links, 10 tree joints (solid thick dark arrows), 3 closure joints (light), one root joint (dashed, others hidden for clarity), and one subframe. Each joint is composed of three sub-transforms, see Figure 7.](image)

Link and Joint Geometries

Our system generates an automatic 3D rendering of any model by representing links as red/green/blue axes triads and joints with the models shown in Figure 7. The resulting “stick figure” pictures are essentially an automatically-generated version of traditional kinematic diagrams. We additionally allow associating arbitrary 3D triangle mesh geometries to each link, loaded from files in standard formats such as VRML97 and COLLADA. These geometries, if present, are used to enhance the visual rendering and are pick-able using the mouse in the interactive interface. We do not currently compute collisions between them or do collision-free path planning, though these are well-studied problems and could be added with appropriate engineering, if desired.

Root Joints

In addition to the joints modeling the kinematic structure of the module, there are also a set of root joints in our system, one per link. Each root joint is a general (unconstrained 6 DoF) joint from its associated link directly to the ground link. Root joints serve several purposes: they act as sentinels for the spanning tree data structure by providing a default parent for any link that may lose its parent during topological mutations, and they can also be made tree joints to emulate Cartesian coordinate modeling (Figure 10).

Joint Mobility Model

Joints are selected from a catalog of 12 different types (Figure 7); though the examples in this paper only use revolute, general (unconstrained 6 Degree-of-Freedom) and fixed (totally constrained 0 DoF), our system can handle all lower-pair joints except helical, and also a selection of higher pairs. Full details of this joint model are given in [16]. Joints are topologically directed such that the joint transform is a rigid-body transform taking coordinates from the frame of the adjacent child link to the frame of the adjacent parent link. As shown at the top of Figure 7, joint transforms are further broken into two positioning transforms, which situate the mobility space of the joint (e.g. the axis of rotation for a revolute joint) relative to the adjacent links, and a mobility transform which is constrained to the subspace of rigid transforms corresponding to the joint type (e.g. the mobility transform of a revolute joint in our system is constrained to the space of rotations about the Z axis, possibly with limits).

The value of the mobility transform is explicitly specified for tree joints, but implied for closures. A composite model transform (CMT) can be directly computed for any link or joint coordinate frame by composing the tree joint transforms from there to the ground link.

Link subframes (Figure 6) may be specified that set positioning transforms for connected joints; these are used in particular to define the poses of inter-module attachment points.

Finally, any joint may be inverted. This essentially swaps its parent and child links and sets a flag that all uses of its mobility transform should be replaced by the inverse transform; though the actual details are more complex because the spanning tree structure must be updated (Algorithm 4, below).

Two-Level Hierarchical Kinematic Graph

So far we have covered the representation for kinematics and geometry within a single module. We extend this to the full multi-module framework by creating a two-hierarchy of nested kinematic graphs, or linkages, as shown in Figure 8. The top-level (outemost) linkage represents the entire multi-module model. The inner nested sublinkages correspond to individual modules. Inter-module connections (and root joints for the inner sublinkages) are the only crossing joints that extend between links in different linkages. Kinematic
Figure 8. Example of a two-level hierarchical kinematic graph with three sublinkages. The model we use in this paper encapsulates each robot module in a sublinkage; inter-module connections are modeled as crossing joints, which are typically fixed (i.e. 0 DoF rigid).

Figure 7. A variety of kinematic joint types can be modeled in our system, though the examples considered in this paper only use revolute, general (unconstrained 6 DoF), and fixed (0 DoF) joints. In this work, fixed joints are used to represent rigid inter-module connections. Each joint is represented as a serial chain of three transforms, as shown at top. The child-to-mobility and mobility-to-parent transforms set the pose of the mobility space (e.g. the axis of rotation for a revolute joint). Overall, the composition of these transforms defines the rigid transform from the coordinate frame of the adjacent child link to the coordinate frame of the adjacent parent link. The directionality of these transforms is optionally shown with cones pointing from child to parent.

spanning trees are still maintained within each sublinkage, and the total flattened graph formed by simply ignoring the sublinkage boundaries also has a well-defined spanning tree.

Reduced vs Cartesian Coordinates

In general there is not a single unique spanning tree for a given kinematic structure; different choices of closure joints imply different spanning trees. This is actually semantically meaningful: the kinematic constraints implied by a tree joint are always guaranteed to be maintained (and require no special computation), but the constraints implied by closure joints must be actively enforced by the system. We use an iterative least-squares numerical solver, a common technique in robotics [15, 16]. This works well in general, but the iterative optimization style of constraint satisfaction means that (a) computation is continually expended to maintain closure joint constraints (in fact, the computational cost is typically quadratic in the number of closures when using the SVD to implement least-squares, as we do); and (b) some residual or transient error is still possible (Figure 9).

Figure 9. Modeling rigid inter-module connections is possible in a Cartesian coordinate model, but can lead to erroneous motions, because the system must constantly try to enforce the rigidity of each connection. This issue can be mitigated in a few ways, but since many practical modular constructions are tree-structured, one way is to convert the Cartesian model of the assembly, which is effectively cyclic, into a reduced coordinate model with no cycles.
The designer who models each module is free to decide how to assign closure joints—and thus how to design the spanning tree—within the module. Similarly, the operator who connects modules together (as in Figure 2) is free to decide whether the added inter-module connection joints should be closures (the default) or tree joints. Conversion from closure to tree and from tree to closure is accomplished by triggering the MakeTree resp. MakeClosure algorithms, given below.

As Figure 10 shows, some terminology has been developed for two particular cases. The case where the spanning tree is “as deep as possible”, i.e., when the number of closure joints is minimized, is called a (fully) reduced coordinate model. The opposite case, where the spanning tree is “as shallow as possible”, is called a Cartesian coordinate model [24]. The root joints in our kinematic representation can be used to build Cartesian coordinate models, as shown in Figure 10.

![Figure 10](image)

**Figure 10.** Top: using root joints to emulate Cartesian coordinate modeling. Each root joint has tree disposition (dark arrows) and sets the pose of the connected link directly with respect to the ground link. All other joints are closures (light arrows). Bottom: the same kinematic topology in a reduced coordinate model. Root joints are still present connecting each link to the ground link even in the reduced coordinate model; however, they are all general (unconstrained 6 DoF) closure joints (to reduce clutter such unconstrained root closures are normally not rendered).

Cartesian-coordinate models are currently popular, and are prevalent in game-oriented physics engines [14]. However, assemblies of modules are often tree structured (no cycles), and it is thus desirable to use MakeTree to build fully reduced coordinate models. Even when cycles are present, only one of the inter-module connections in each cycle need remain a closure joint.

## 5. Algorithms for Topological Evolution

The stage is now finally set to present the topological evolution algorithms. Adding new modules is nearly trivial: simply instantiate a copy of the appropriate module’s kinematic graph. The initial kinematic configuration and global pose of the new module can be set in various ways, including simply letting the user place it in the scene with the mouse. Adding and removing inter-module connections is also straightforward so long as they are represented as closure joints, because this does not change the topology of any existing spanning tree. The iterative least-squares solver can maintain these closure constraints—as long as there is no incompatible overconstraint—but again there can be issues as in Figure 9, in addition to the computational overhead.

**MakeTree**, Algorithm 1, is the first interesting algorithm (see Tables 1 and 2 for explanation of the notation). The operator can engage MakeTree to convert any existing closure joint into a tree joint, thereby changing the spanning tree topology. A call to MakeTree on a closure joint \( j \) first makes the prior tree parent of \( j \)'s child link \( c \) a closure, and then replaces the tree parent of \( c \) with \( j \). \( j \)'s mobility transform is clamped to the allowed mobility space and limits (which matters if \( j \) was previously a broken closure).

### Algorithm 1: MakeTree

```plaintext
for i ← p_j to g_0 do
    if i = c_j then error C_j is an ancestor of p_j else i ← i_p_r
    if Crossing?(j) then MakeGround(c_j)
    else if (c_j = g_l) then MakeGround(p_j)
    M_f_j ← CLAMP((CMT(p_m_j)^(-1))CMT(c_m_j))^{th_i}, j)
    M_{p_c_j} ← θ (previous parent of c_j becomes a closure),
    p_{c_j} ← j
```

Unless \( c \) is the ground link or \( j \) is crossing, which are special cases, MakeTree is \( O(h) \) with \( h \) the maximum spanning tree height of the parent and child links of \( j \) (the limiting computation is actually computing the CMTs). When \( c \) is a ground link or \( j \) is a crossing joint a re-grounding is triggered via MakeGround, Algorithm 3, and in that case the time complexity of MakeTree is dominated by the call to MakeGround.

**MakeClosure**, Algorithm 2, is the complimentary operation to MakeTree, and converts a joint currently in the spanning tree into a closure. The child link is re-parented to its root joint, whose mobility transform is updated so that the link does not change global pose due to the change in topology. MakeClosure is \( O(h) \) where \( h \) is the spanning tree
height of the child link, again to compute the CMT.

**Algorithm 2: MAKECLOSERUH(tree joint \( j \))**
\[
\begin{align*}
M_{r_{cj}} &\leftarrow \text{CMT}(c_j) \\
M_j &\leftarrow \emptyset \text{ (closure joint mobility transform is implicit)} \\
p_{c_j} &\leftarrow r_{cj} \text{ (child link is reparented to its root joint)}
\end{align*}
\]

MAKEGROUND is not only a subroutine called by some special cases in \textsc{MakeTree}, but also may be called directly by the operator to manually change the ground link of a linkage. \textsc{MakeGround} in works together with \textsc{INVERT}, Algorithm 4, which reverses both the topological and kinematic semantics of a joint. \textsc{INVERT} may also be called directly by the operator.

**Algorithm 3: MAKEGROUND(link \( k \) in linkage \( L \))**
\[
\begin{align*}
\text{if} (k \neq g_L) &\text{ then return} \\
\text{if} \ \text{ROOTLINK}?(k) &\text{ then} \\
&\text{foreach root joint} \ j \ in \ L \ do \\
&\text{if} \ \text{TREE}?(j) \text{ then } M_j \leftarrow M_{r_k}^{-1} M_j \\
&\text{let} \ r \ be \ a \ new \ root \ joint \ for \ the \ old \ ground \ link \ of \ L \\
&\quad M_r \leftarrow M_{r_k}^{-1} \\
&\quad p_{g_L} \leftarrow p_{g_L}, p_L \leftarrow r \\
&\quad \text{add} \ r \ to \ L, \ \text{remove old root joint} \ r_{g_L} \ \text{from} \ L \\
&\quad r_{g_L} \leftarrow r, g_L \leftarrow k \\
&\text{if} \ k = g_0 \text{ then remove} \ r_k \ \text{from} \ L \ (\text{top-level gnd, no root}) \\
&\text{else} \ \text{INVERT}(p_k) \ (\text{invert parent joint})
\end{align*}
\]

\textsc{MakeGround} has two cases. The shorter one is when the new ground link \( l \) is not currently a root link, i.e. is not parented directly to the current ground link via its root joint. In this case \textsc{MakeGround} actually defers to \textsc{Invert} on the parent joint of \( l \). Though \textsc{Invert} will itself call back to \textsc{MakeGround}, there is no circularity because at the time of this re-entrant call, \( l \) will always be a root link. The time complexity in this case is the same as that of \textsc{Invert} (tree joint case) on the parent joint.

Making a current root link \( l \) the ground link involves three steps. First, the root joints of all other links are re-parented to \( l \). Second, a root joint is connected from the prior ground link to the new ground. Third, the root joint for \( l \) is re-attached wherever the old ground link’s root joint was attached. The time complexity in this case is \( O(|K|) \), where \( K \) is the set of links, because all their root joints need to be reparented.

\textsc{Invert} flips the topology of a joint in-place, so that its prior parent becomes its new child, and vice-versa. For a closure joint this is a local procedure: the positioning transforms are swapped and inverted, the parent and child links are swapped, and the mobility inversion flag is flipped. The time complexity in this case is \( O(1) \). For a tree joint, inversion triggers a re-grounding—the child link \( c \) of the joint to be inverted becomes the new ground link, and all the other joints on the tree path from \( c \) to the prior ground link are also inverted. For simplicity, inversion is not supported for the case of crossing tree joints. The time complexity for inverting a tree joint is \( O(h + |K|) \) where \( h \) is the spanning tree height of the parent of the joint to be inverted and \( K \) is the set of links. The \( O(|K|) \) term is due to the call to \textsc{MakeGround}, at which point \( c \) will always be a root link.

\[
\begin{align*}
\text{Algorithm 4: INVERT(non-root joint} \ j) &
\end{align*}
\]

\textsc{MakeGround} has two cases. The shorter one is when the new ground link \( l \) is not currently a root link, i.e. is not parented directly to the current ground link via its root joint. In this case \textsc{MakeGround} actually defers to \textsc{Invert} on the parent joint of \( l \). Though \textsc{Invert} will itself call back to \textsc{MakeGround}, there is no circularity because at the time of this re-entrant call, \( l \) will always be a root link. The time complexity in this case is the same as that of \textsc{Invert} (tree joint case) on the parent joint.

Making a current root link \( l \) the ground link involves three steps. First, the root joints of all other links are re-parented to \( l \). Second, a root joint is connected from the prior ground link to the new ground. Third, the root joint for \( l \) is re-attached wherever the old ground link’s root joint was attached. The time complexity in this case is \( O(|K|) \), where \( K \) is the set of links, because all their root joints need to be reparented.

\textsc{Invert} flips the topology of a joint in-place, so that its prior parent becomes its new child, and vice-versa. For a closure joint this is a local procedure: the positioning transforms are swapped and inverted, the parent and child links are swapped, and the mobility inversion flag is flipped. The time complexity in this case is \( O(1) \). For a tree joint, inversion triggers a re-grounding—the child link \( c \) of the joint to be inverted becomes the new ground link, and all the other joints on the tree path from \( c \) to the prior ground link are also inverted. For simplicity, inversion is not supported for the case of crossing tree joints. The time complexity for inverting a tree joint is \( O(h + |K|) \) where \( h \) is the spanning tree height of the parent of the joint to be inverted and \( K \) is the set of links. The \( O(|K|) \) term is due to the call to \textsc{MakeGround}, at which point \( c \) will always be a root link.

**Table 1. Auxiliary functions used in algorithms.**

- **CROSSING(joint \( j \))** check if \( j \) crosses a linkage boundary
- **TREE?(joint \( j \))** check if \( j \) is a tree joint, i.e. if \( p_{c_j} = j \)
- **CLOSURE?(joint \( j \))** check if \( j \) is a closure joint, i.e. if \( p_{c_j} \neq j \)
- **ROOTLINK?(link \( k \))** check if \( k \) is a root link, i.e. if \( p_k = r_k \)
- **CMT(frame \( f \))** get the composite model transform from \( f \) to \( g_0 \)
- **CLAMP(transform \( M \), joint \( j \))** clamp \( M \) to the mobility space of \( j \)

**6. CONCLUSIONS AND FUTURE WORK**

We have presented implemented algorithms and applications examples for a new graphical operations interface specifically for articulated modular robots. Such robots have been studied by robotics researchers for several decades, and are now making their way into space exploration practice, with systems such as JPL’s new TriATHLETE. We demonstrated how our interface could be used by an operator to quickly develop mission operations both for canonical and novel multi-module
Funding for this work was provided through the Director’s Research and Development Fund at NASA/JPL/Caltech. The TriATHLETE graphical model shown in figures was developed by Mark Powell at NASA/JPL/Caltech.

ACKNOWLEDGMENTS

Funding for this work was provided through the Director’s Research and Development Fund at NASA/JPL/Caltech. The TriATHLETE graphical model shown in figures was developed by Mark Powell at NASA/JPL/Caltech.

REFERENCES


**BIOGRAPHY**

Marsette A. Vona is a postdoctoral associate at, and a recent Ph.D. graduate of, the Computer Science and Artificial intelligence lab at Massachusetts Institute of Technology. While a graduate student, he took a two year leave of absence to work at JPL on the science operations software for the Spirit and Opportunity Mars rovers, for which he shared the NASA Software of the Year Award in 2004. Marsette’s research interests lie broadly in computation that interacts with the physical world, including robotics, user interfaces, geometric design systems, and data visualization.