Hierarchical Decomposition and Kinematic Abstraction with Virtual Articulations

Marsette Vona

Northeastern University College of Computer and Information Science, Boston, Massachusetts, e-mail: vona@ccs.neu.edu

Abstract. We introduce a novel hierarchical model to partition a kinematic system into a set of nested subsystems. This is framed in a *mixed real/virtual* context, where some joints and links may exist in simulation only. We then use this capability to build a precise form of *kinematic abstraction*, where a potentially complex subsystem can be virtually replaced by a simpler "interface." Hierarchy and abstraction are interesting because they can help manage complexity in large (100+ DoF) mixed real/virtual mechanisms. We prove that checking if an abstraction is *proper* is PSPACE-hard, but show that even improper abstractions can be useful. Topological algorithms are presented for decomposing a hierarchical or abstracted kinematic system into subsystems that can be treated in isolation, thus speeding up kinematic computations. We demonstrate on a simulation of a hybrid serial/parallel modular tower with over 100 revolute joints.

Key words: kinematic graphs, virtual joints, topological decomposition

1 Introduction

Abstractions, where a complex implementation is hidden behind a simpler interface, are well known for managing complexity in computation. Is there any correlate in the domain of kinematics? We demonstrate the affirmative by introducing *structure abstraction* (Sec. 3). This novel and concrete technique can be used to effectively hide a complex kinematic subsystem behind a simpler "interface mechanism." Our approach is built on two foundations: *mixed real/virtual* models, where some links and joints are virtual; and *hierarchical linkages*[1]. By closing chains, virtual elements can be used to specify constraints (Fig. 3 gives an example). They also make structure abstraction more practical, as the interface mechanism can be virtual. We introduce hierarchical linkages, the second foundational technique, in Sec. 2.

Hierarchy and abstraction are interesting because they can help manage complexity in large systems, just as in other domains of computing and engineering. In particular, we are interested in high-DoF redundant mechanisms with 100+ DoF. Our original motivation arose from the study of kinematic simulation and control of

[1] We use "linkage" in this work to mean any kinematic system composed of links and joints.

linkage-type modular robots, where many modules can assemble in arbitrary topologies. One use for abstraction here is to help design virtual kinematic constraints—as chain-closing virtual joints—which guide a desired motion. Constraints can be defined for each subsystem and then placed below abstraction barriers. This is particularly useful given that high-DoF constructions often contain repeated substructure.

Abstractions can also lead to faster kinematic computation, e.g. numeric IK, by treating interface mechanisms as stand-in replacements for their more complex "implementations." The resulting motion can then be imposed on each implementation *in isolation*. We give the necessary algorithms in Sec. 4 to partition a hierarchical kinematic graph according to both (1) its biconnected components (a well-known decomposition) and (2) the imposed hierarchy. The latter ensures that interfaces are solved before implementations, as desired.

We implemented these algorithms as part of a general mixed real/virtual spatial kinematic simulation environment [1]. Sec. 5 demonstrates an example of interactive IK for a hybrid serial/parallel tower structure with over 100 revolute joints.

We find relatively little prior work in the area of kinematic abstraction. Davis [2] explored the idea of geometric abstraction, including a one-paragraph mention of "kinematic device as black box." Zanganeh and Angeles [3] studied partitioning of topologically large kinematic graphs, but did not separate interface from implementation. In [4], Williams and Mahew presented a tower structure similar to ours, but smaller in topological scale. More significantly, they developed an IK solving optimization related to structure abstraction, but only for one hand-decomposed instance. Our algorithms apply in the general case with no manual intervention.

2 Hierarchical Linkages

Fig. 1 shows a kinematic graph L where the vertices are links (rigid bodies) and the edges are joints. Our implementation supports 12 different joint types. Cycles (closed chains) are allowed, but we always identify a spanning tree and distinct *closure* joints. The *parent* link of a joint is generally the one closer to the tree root. We allow the designer to identify this spanning tree by marking one closure joint in each cycle; it could also be automatically found. The pose of each link l is considered to be defined by the path of joint transforms from l to the tree root.

Such a flat graph L can be turned into a *hierarchical kinematic graph* $\mathscr{L}$ by imposing a properly nested set of disjoint edge cuts as shown in Fig. 2. We call each such demarcated subgraph a *sublinkage*. Further, we propose that a *disposition* of "driving," "driven," or "simultaneous" is assigned to each sublinkage. If P is the *parent* sublinkage immediately enclosing *child* sublinkage C, the disposition of C determines whether it is considered rigid with respect to P (driving), vice-versa (driven), or whether they are both mobile in the same context (simultaneous).

For real linkages, such relationships may only be enforceable if fully actuated. For mixed real/virtual models, we can assume all virtual joints are actuatable; the real joints can be driven accordingly, again assuming no physical underactuation.

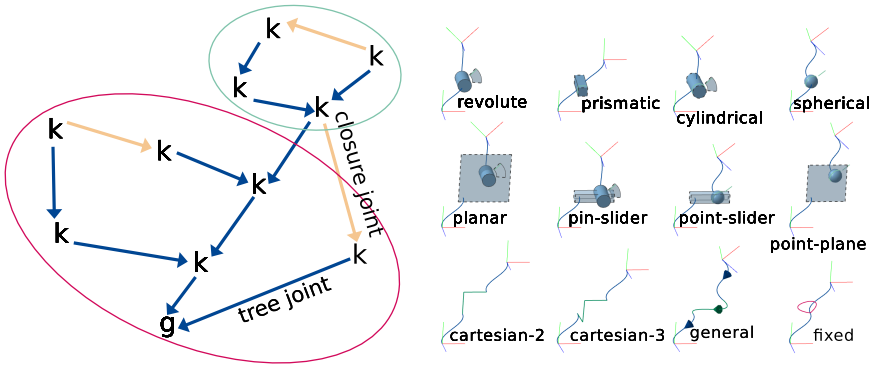

Fig. 1 The edges in a *kinematic graph* (left) correspond to joints and the vertices to links. We allow closed chains but always distinguish a spanning tree (dark edges). 12 spatial joint types are available in our implementation (right), including all lower pairs except helical. Strongly connected components (circled, edge direction ignored) in the graph can be solved independently.

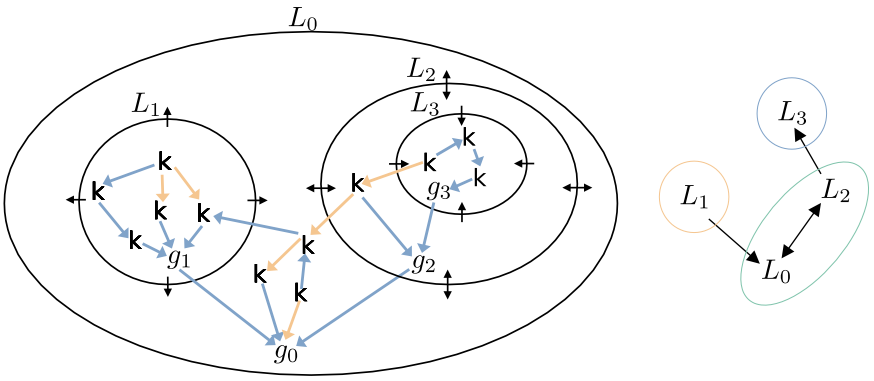

Fig. 2 To add hierarchy, partition a kinematic graph into a proper nesting of subgraphs. Each is specified as *driving* (e.g. L_1), *driven* (L_3), or *simultaneous* (L_2). The strongly connected components (circled) of a directed meta-graph of the subgraphs (right) are solvable independently.

We do not actually remove the *crossing* joints in the edge cuts defining $\mathscr{L}$. Since the edge cuts are all disjoint, any crossing joint is cut by exactly one sublinkage boundary. Again assuming P to be the parent sublinkage of C, an *outcrossing* joint connects its child link in C to a parent link in P, and vice-versa for an *incrossing* joint. In the algorithms it is sometimes necessary find the innermost sublinkage containing a given joint or link. This is unambiguous except for crossing joints; we simply define them to be members of the inner of the crossed sublinkages.

Two additional topological restrictions are also imposed on crossing tree joints (there are no constraints on crossing closures): First, each sublinkage C always has exactly one outcrossing tree joint (except at the top level) connecting the root of its spanning tree to a link in the parent sublinkage P. This ensures that each sublinkage

has its own well-defined spanning tree. Second, incrossing tree joints are disallowed for a driven sublinkage C. Otherwise, changing the relative pose of links in C could change the relative pose of other links in the parent sublinkage P, violating the specified semantics that P should be rigid with respect to C.

3 Structure Abstraction

Structure abstraction is achieved by (a) encapsulating a connected part A of a linkage L s.t. A becomes demarcated as a simultaneous sublinkage of of L, and (b) substituting a virtual linkage I for A in L, with I simultaneous in L and A a driven sub-linkage of I. Fig. 3 shows an example. The concept parallels traditional abstraction in computing: I can be simpler than A, but it should capture all of the behavior of A that would be relevant to the surrounding mechanism.

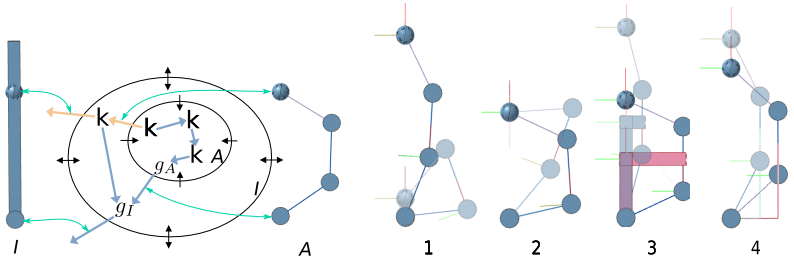

Fig. 3 In *structure abstraction* (left), an *implementation* linkage A is made a driven sublinkage of a new virtual *interface* linkage I. I replaces A in-context. The primary motion of interest in this example is the end-to-end stretching of A (1). Details, such as virtual prismatic joints (3) added to constrain an unwanted postural freedom (2), can be hidden underneath I. (4) shows a shorthand using a *Cartesian-2* joint that achieves the same effect as the virtual prismatic assembly in (3).

Making A a driven sub-linkage of I is not the only possible way to define abstraction in kinematics. For example, keeping A simultaneous could also make sense. However, the design choice to make A driven creates a fairly strong form of abstraction: motion of L, with I substituting for A, is independent of the motion of A. This can be helpful both (1) as a simplifying design aid (e.g. to allow virtual constraints as in Fig. 3 to to be abstracted away) and (2) to allow decomposition of the overall kinematic system into independently solvable subsystems, e.g. for IK computation.

This decoupling power comes with a trade-off: there is no built-in constraint to ensure that A can reach every configuration to which it may be driven by I. We say that I is a *proper abstraction* of A if all of the reachable configurations of I, when embedded in the surrounding linkage L, drive reachable configurations of A. It would be desirable to have an efficient algorithm that could determine, for any L,

I, and A, whether I is a proper abstraction. This may be possible in some special cases, but unfortunately, the general case is easily shown to be hard.

Theorem 1 *Determining whether an abstraction is proper is PSPACE-hard.*

Proof By reduction from the *reachability problem* for 2D revolute linkages. Kempe's theorem [5] establishes that I could be constructed s.t. the links which drive A could move to arbitrary poses. But the problem of determining the reachability of arbitrary configurations of A from a starting configuration is known to be PSPACE-hard even for the restricted case of 2D revolute linkages [6, 7].

Thus, if a proper abstraction is required, it is up to the designer of that abstraction to ensure it. Sometimes this is easily done by construction. However, even improper abstractions can be useful when combined with task-priority IK (e.g. [8])—if the kinematic constraints in A are given higher priority than those of, say, the crossing joints between I and A, then the motion of A will remain feasible. It will not exactly match the driving links in I, but because they are virtual, this can be acceptable.

4 Decomposition Algorithms

Practical algorithms for simulation and kinematic control of arbitrary topology linkages have polynomial runtime per iteration; e.g. least-squares IK with the SVD is typically quadratic in the number of DoF [1]. Thus it can be faster to run algorithms on disjoint subsystems, provided that their motion is actually independent. The overall motion is then the union of the motions of all sublinkages.

One well-known decomposition [9] finds biconnected components of the kinematic graph (Fig. 1), on the intuition that overlapping closed chains must be solved simultaneously. Our hierarchical model permits this, but also imposes additional lines of decomposition. Specifically, a driving sublinkage is solved before its parent, and vice-versa for a driven sublinkage. In fact, this ordering is one way to *enforce* the imposed driving/driven relationship. Because we also allow simultaneous sublinkages, this decomposition is not necessarily the same as the edge cuts defining the hierarchy. We must rather consider the strongly connected components of a directed meta-graph whose vertices correspond to the sublinkages and whose edges are directed according to sublinkage disposition. Fig. 2 shows an example. (Note that a simultaneous sublinkage is connected to its parent with a bi-directional edge.)

In this section we present algorithms which decompose a given hierarchical linkage $\mathscr{L}$ into a set of disjoint components respecting both the biconnected components of the *flattening* of $\mathscr{L}$ (i.e. the underlying kinematic graph disregarding edge cuts) and also the strongly connected components of the associated meta-graph. The top-level call, DECOMPOSE (Alg. 1), has three phases. First, ASSIGNROUNDS (Alg. 2), labels the strongly connected components in $\mathscr{L}$ and also assigns a *solve round* to every SCC in $O(|\mathscr{L}|)$. (SCC and solve round labels are considered "inherited" by both the sublinkages within an SCC as well as the individual joints within them.) A sort by increasing round, performed at the end of DECOMPOSE in $O(|\mathscr{L}|\log|\mathscr{L}|)$, gives a solve ordering respecting all driving/driven relations.

The second phase of DECOMPOSE calls TRACESUPPORTS (Alg. 5) to identify the set of tree joints whose motion can affect the state of each closure joint. TRACESUPPORTS is $O(|T \cup C| + |T||C|)$ where T is the set of tree joints and C the set of closures in the flattening of $\mathscr{L}$. The helper function LOCKEDWRT(t,c) (Alg. 6) is key here, as it will ensure that tree joints will be considered locked with respect to closures unless both are in the same SCC.

The final phase calls FINDCOMPONENT (Alg. 7) to extract independent components by tracing transitive overlaps of closure supports. This phase is $O(|T||C|)$ because the $O(|T|)$ support of each closure is traversed once, and likewise for the $O(|C|)$ set of *supported-closures* for each of the $O(|T|)$ supporting tree joints.

Algorithm 1: DECOMPOSE($\mathscr{L}$)

Input: hierarchical linkage $\mathscr{L}$ with top-level sub-linkage L_0
Output: ordered partition $\mathscr{C}$ of closure joints
corresponding ordered set $\mathscr{T}$ of sets of unlocked supporting tree joints
unlocked support chains $S{\downarrow_c}, S{\uparrow_c}$ for each closure
ASSIGNROUNDS($\mathscr{L}$)
let T, C be the tree/closure partition of the joints in the flattening of $\mathscr{L}$
foreach $j \in C$ **do** $S{\downarrow_c} \leftarrow \emptyset$, $S{\uparrow_c} \leftarrow \emptyset$, mark j unassigned
foreach $j \in T$ **do** let *supported-closures*$_j \leftarrow \emptyset$, mark j unassigned
let $U \leftarrow \emptyset$ ▷*the closures will be collected here*
$U \leftarrow$ TRACESUPPORTS(g_0, U) ▷g_0 *is the ground link in* L_0
let $\mathscr{C} \leftarrow \emptyset$, $\mathscr{T} \leftarrow \emptyset$
while $U \neq \emptyset$ **do**
  let c be the first element of U, $C \leftarrow \emptyset$, $T \leftarrow \emptyset$
  FINDCOMPONENT(c, U, C, T)
  add (C, T) to $(\mathscr{C}, \mathscr{T})$
sort $(C, T) \in (\mathscr{C}, \mathscr{T})$ in order of increasing solve round of C
return $(\mathscr{C}, \mathscr{T})$

Algorithm 2: ASSIGNROUNDS($\mathscr{L}$)

Input: hierarchical linkage $\mathscr{L}$ with top-level sub-linkage L_0
Output: each sub-linkage is marked with its solve round and its SCC in $\mathscr{L}$
foreach sub-linkage $L \in \mathscr{L}$ **do** $round_L \leftarrow -1$, $scc_L \leftarrow -1$
let set of source sub-linkages $U \leftarrow \emptyset$, next SCC id $n \leftarrow 0$
FINDSOURCESFROM(L_0, U)
foreach $L \in U$ **do** $n \leftarrow$ ASSIGNROUNDSFROM$(L, 0, n, n+1)$

The full DECOMPOSE algorithm is thus $O(|T \cup C| + |T||C| + |\mathscr{L}| \log |\mathscr{L}|)$. A classical result in graph theory [10] is that biconnected components can be found in $O(|T \cup C|)$; DECOMPOSE is asymptotically slower, but also supports hierarchical decomposition. Also, since many iterative numerical algorithms are at least quadratic in the number of DoF (which is at worst proportional to $|T|$), DECOMPOSE does not typically increase the overall computational complexity.

Algorithm 3: FINDSOURCESFROM(L,U)

Input: sub-linkage L, collected source sub-linkages U
Output: true if L drives its parent, updated U
let $dbd \leftarrow$ false ▷*driven by descendant*
foreach child sub-linkage M of L **do**
  if FINDSOURCESFROM(M,U) **then** $dbd \leftarrow$ true
if $\neg dbd$ and (($L = L_0$) or (L is driving)) **then** add L to U
return (L is driving) or (dbd and (L is simultaneous))

Algorithm 4: ASSIGNROUNDSFROM(L,r,i,n)

Input: sub-linkage L, solve round r, id i of SCC containing L, next unused SCC id n
Output: the new next unused SCC id
$round_L \leftarrow r$, $scc_L \leftarrow i$
foreach child sub-linkage M of L **do**
  if ($round_M < 0$) and (M not driving) **then**
    if M driven **then** $n \leftarrow$ ASSIGNROUNDSFROM$(M, r+1, n, n+1)$
    else $n \leftarrow$ ASSIGNROUNDSFROM(M,r,i,n) ▷*M is simultaneous*
if ($L \neq L_0$) and ($round_{parent_L} < 0$) and (L not driven) **then**
  if L driving **then** $n \leftarrow$ ASSIGNROUNDSFROM$(parent_L, r+1, n, n+1)$
  else $n \leftarrow$ ASSIGNROUNDSFROM$(parent_L, r, i, n+1)$ ▷*L is simultaneous*
return n

Algorithm 5: TRACESUPPORTS(l,U)

Input: start link l, collected closures U
Output: U updated with newly found closures, and support chains for such
foreach j s.t. $parent_j = l$ **do**
  let $breadcrumb_l \leftarrow j$, $i \leftarrow child_j$
  if TREE?(j) **then** $U \leftarrow$ TRACESUPPORTS(i,U)
  else ▷ j is a closure joint
    add j to U
    repeat ▷ *trace down to least common ancestor*
      $p \leftarrow parent_i$, $i \leftarrow parent_p$
      if ¬LOCKEDWRT?(p,j) **then** append p to $S{\downarrow_j}$, add j to $supported\text{-}closures_p$
    until $breadcrumb_i \neq \emptyset$
    ▷*i is now the LCA of $child_j$ and $parent_j$*
    repeat ▷ *trace up from LCA*
      $c \leftarrow breadcrumb_i$, $i \leftarrow child_c$
      if ¬LOCKEDWRT?(c,j) **then** append c to $S{\uparrow_j}$, add j to $supported\text{-}closures_c$
    until $c = j$
$breadcrumb_l \leftarrow \emptyset$
return U, support chains as side-effect

Algorithm 6: LOCKEDWRT?(t,c)

Input: tree joint t in sub-linkage L_t, closure joint c in sub-linkage L_c
Output: whether t is to be considered locked with respect to c
if t is explicitly locked **then return** true
let $i \leftarrow scc_{L_c}$
if CROSSING?(c) and (L_c driving) **then** $i \leftarrow scc_{parent_{L_c}}$
return ($i \neq scc_{L_t}$)

Algorithm 7: FINDCOMPONENT(c, U, C_i, T_i)

Input: closure c to add to C_i, unexplored closures U
sets C_i and T_i of coupled closures and supporting unlocked tree joints
Output: updated U, C_i, and T_i
remove c from U, add c to C_i, mark c assigned
foreach unlocked supporting tree joint t in $(S{\downarrow_c}, S{\uparrow_c})$ **do**
  if t unassigned **then**
    add t to T_i, mark t assigned
    foreach unassigned closure u in *supported-closures*$_t$ **do** FINDCOMPONENT(u, U, C_i, T_i)

5 Scaling Results

We have implemented these algorithms as part of a new mixed real/virtual spatial kinematic simulator [1]. This environment supports general open- and closed-chain models using joints selected from the catalog in Fig. 1, and includes an interactive constraint solver based on task-priority iterative damped least-squares. Model topology can change on-line, and DECOMPOSE is automatically invoked as necessary.

In this section we demonstrate the scalability of our approach by showing interactive IK control of a simulated hybrid serial/parallel tower with 120 revolute joints and over 150 additional virtual joints (Fig. 4). Two layers of structure abstraction are applied, breaking up the operator's motion specification task and also enabling hierarchical decomposition to speed IK solving. We initially explored such a tower in [11], but that work was hand-coded and did not use the DECOMPOSE algorithm.

The tower is constructed of a chain of self-similar blocks. The actual joints comprising each block are redundant and can move in a variety of ways, but the operator intends only a subset of this motion. Extrinsically, the block should only have two DoF: it should be able to tilt left and right and to expand up and down. This forms the highest-level abstraction (**C** in the figure). Within this, a secondary constraint is that each 4-bar leg should effectively act like a piston, with the middle link remaining parallel to the axis of the piston. Virtual joints are added to enforce this local posture constraint, and then abstracted below a 3DoF RPR virtual interface (**B**). Fig. 3 shows the same construction. In this case, the abstractions can be kept proper by limiting the range of motion of the joints in the interfaces **B** and **C**.

These particular motion constraints, and this particular set of abstractions, are merely the designs of the operator. Other constraints and abstractions are possible: the idea is that the operator may express a desired set of motion constraints by designing constraints and structure abstractions.

Blocks can be strung together for towers of varying height. This final assembly is done at abstraction level **C**, so that the top-level linkage is simply a linear chain along the "backbone" of the tower (c.f. [12]). The operator can then interactively specify a motion, e.g. by click-and-drag interaction, for any **C**-level link or joint.

To check the speedup afforded by hierarchical decomposition, we conducted an experiment with towers of varying heights (up to 15 blocks). Each was tested either as a flat linkage or structured with the above two levels of abstraction. The resulting

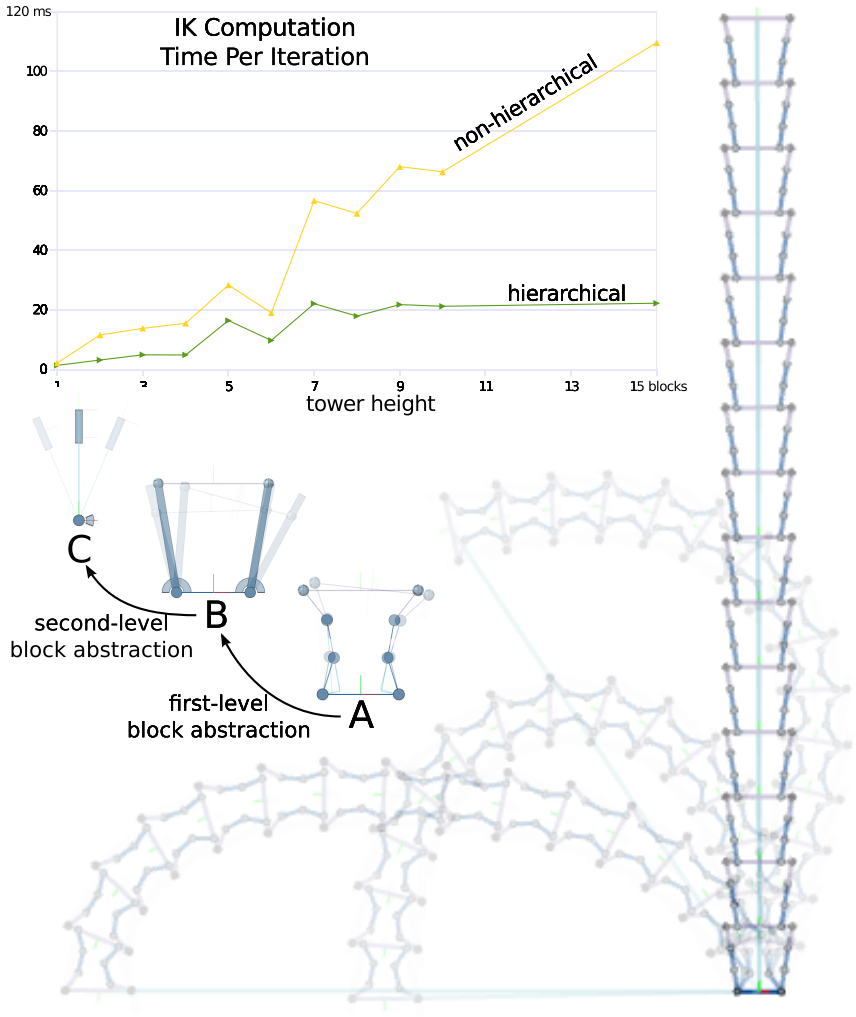

Fig. 4 Simulated interactive IK of a large (> 100 joint) hybrid serial/parallel tower is greatly accelerated by the introduction of two levels of abstraction (and hence hierarchy) for each block. The block legs **A** are virtually constrained and abstracted as in Fig. 3, producing an intermediate virtual model **B**. An additional level of abstraction covers **B** as a limited-travel RP chain **C**. An operator may drag any joint or link in the "backbone" chain of level-**C** interface mechanisms; the system interactively computes a corresponding motion for the tower respecting all constraints.

measured computation times (on a typically loaded workstation) are comparatively plotted in Fig. 4. The hierarchical models remain at about 20ms per iteration, which is acceptable for interactive response. But performance degrades to over 100ms for a 15 block non-hierarchical tower, which results in very sluggish behavior.

These timings of course depend on the speed of the workstation. Furthermore, the hierarchical decomposition does not change the asymptotic cost of the IK computation, which is still quadratic in the number of joints solved in any single system (and the number of DoF in the level-**C** backbone still scales linearly with tower height). But lower constant factors for the hierarchical case mean that larger systems can be handled in practice before reaching the limits of interactivity.

6 Conclusions

In this paper we introduced a new hierarchical way to structure mixed real/virtual kinematic systems, and used it to define a novel form of abstraction for kinematics. As in other areas of computing and engineering, hierarchy and abstraction can help deal with large systems. We demonstrated our approach in a simulation of a tower structure with over 100 DoF. However, our algorithms and their implementation [1] are general across a broad class of spatial open- and closed-chain mechanisms.

Acknowledgements This work was performed at MIT, and was funded under the NSF EFRI program. Additional funding was provided under a Caltech/NASA/JPL DRDF grant.

References

1. M. A. Vona. *Virtual Articulation and Kinematic Abstraction in Robotics*. PhD thesis, EECS, Massachusetts Institute of Technology, August 2009.
2. E. Davis. Approximation and abstraction in solid object kinematics. Technical Report TR1995-706, New York University, Department of Computer Science, 1995.
3. K. E. Zanganeh and J. Angeles. A formalism for the analysis and design of modular kinematic structures. *The International Journal of Robotics Research*, 17(7):720–730, 1998.
4. R. L. Williams and J. B. Mayhew. Control of truss-based manipulators using virtual serial models. In *The 1996 ASME Design Eng. Tech. Conf. and Comp. in Eng. Conf.*, 1996.
5. A. B. Kempe. On a general method of describing plane curves of the *n*th degree by linkwork. *Proceedings of the London Mathematical Society*, 7:213–216, 1876.
6. R. Connelly and E. D. Demaine. Geometry and topology of polygonal linkages. In Jacob E. Goodman and Joseph OŔourke, editors, *CRC Handbook of Discrete and Computational Geometry*, chapter 9, pages 197–218. CRC Press, 2nd edition, 2004.
7. J. Hopcroft, D. Joseph, and S. Whitesides. Movement problems for 2-dimensional linkages. *SIAM Journal of Computing*, 14:315–333, 1985.
8. P. Baerlocher and R. Boulic. An inverse kinematics architecture enforcing an arbitrary number of strict priority levels. *The Visual Computer*, 20:402–417, 2004.
9. C. Welman. Inverse kinematics and geometric constraints for articulated figure manipulation. Master's thesis, Simon Fraser University, 1993.
10. R. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comp.*, 1:146–160, 1972.
11. C. Detweiler, M. Vona, K. Kotay, and D. Rus. Hierarchical control for self-assembling mobile trusses with passive and active links. In *IEEE International Conference on Robotics and Automation*, pages 1483–1490, 2006.
12. G. S. Chirikjian and J. W. Burdick. The kinematics of hyper-redundant robot locomotion. *IEEE Transactions on Robotics and Automation*, 11(6):781–793, 1995.