Robotica (2011) volume 29, pp. 35–57. © Cambridge University Press 2011
doi:10.1017/S0263574710000664

Robust and reversible execution of self-reconfiguration sequences†

Ulrik Schultz*, Mirko Bordignon and Kasper Stoy

Modular Robotics Lab, Maersk Mc-Kinney Moller Institute, Faculty of Engineering, University of Southern Denmark, Denmark

(Received in Final Form: October 22, 2010)

SUMMARY

Modular, self-reconfigurable robots are robotic systems that can change their own shape by autonomously rearranging the physical modules from which they are built. In this work, we are interested in how to distributedly execute a specified self-reconfiguration sequence. The sequence is specified using a simple and centralized scripting language, which either could be the outcome of a planner or be hand-coded. The distributed controller generated from this language allows for parallel self-reconfiguration steps and is highly robust to communication errors and loss of local state due to software failures. Furthermore, the self-reconfiguration sequence can automatically be reversed, if desired. We verify our approach and demonstrate its robustness in experiments using physical and the simulated ATRON modules, as well as simulated M-TRAN modules. Overall, the contribution of this work is the combination of the tractability of a centralized scripting language with the robustness and parallelism of distributed controllers in modular robots.

KEYWORDS: Modular Robots; Control of Robotic Systems; Robotic Self-Diagnosis and Self-Repair; Robotic Self-Replication; Sensor Networks.

1. Introduction

Modular robotics is an approach to the design, construction, and operation of robotic devices aiming to achieve flexibility and reliability by using a reconfigurable assembly of simple subsystems.[33] Robots built from modular components can potentially overcome the limitations of traditional fixed morphology systems because they are able to rearrange modules automatically on a need basis, a process known as self-reconfiguration, and are able to replace unserviceable modules without disrupting the system's operations significantly. Another potential advantage is the cost-effectiveness over more conventional robots, achieved through large-scale production of simpler and identical modules. In order to realize this potential, the cost of mechatronics has to be kept low and as a consequence we favor simple algorithms that can be executed on resource-constrained embedded systems.

Control of self-reconfiguration can broadly be divided into centralized and distributed approaches. The distributed approaches are considered superior compared to centralized approaches due to their robustness and inherent parallelism, but are, on the other hand, often intractable in terms of controller design. Centralized approaches are more tractable, but have limited robustness because of having a single point of failure. The question we therefore try to answer is if we can bridge the gap between a centralized representation and a distributed controller and thus harvest the advantages of both approaches.

The generation of a centralized representation of a self-reconfiguration sequence has been the focus of many papers[10,12,22,25–27,30–32,36,37] but is not the subject of this paper. Rather, we focus on the transformation of this representation into a distributed controller. This transformation is in principle easy to do, but in practice many difficulties arise: real-world modules may have partial failures in their neighbor-to-neighbor communication abilities and may spuriously fail during the self-reconfiguration sequence. The challenge of making a distributed controller that is robust to these difficulties is significant, but it becomes important to address this challenge as modular, self-reconfigurable robotic systems mature and gradually move into real-world applications — a process stimulated by events such as the ICRA 2008 and 2010 Planetary Contingency Challenge events. Even without these difficulties the actual distributed implementation of the self-reconfiguration sequence is intractable because optimizations, such as the parallelization of operations, are nontrivial and significantly complicate the implementation because a partial ordering of self-reconfiguration operations must be maintained.[6]

The contribution of our work is that we demonstrate how a centralized script can be executed by unreliable hardware and consequently makes it easier for a programmer to specify a self-reconfiguration sequence. To this end, we have implemented a scripting language that can execute self-reconfiguration sequences on the ATRON modular robot[24] distributedly. This scripting language is an extension to the DynaRole language,[7] but provides a number of significant improvements. First, self-reconfiguration sequences are compiled to a robust and efficient implementation based on a distributed state machine. Second, dependencies between operations are explicitly stated to allow independent operations to be performed in parallel while enforcing sequential ordering between actions that are physically dependent on each other. Third, the language is *reversible* meaning that for any self-reconfiguration sequence the reverse one is automatically generated. Any self-reconfiguration process described in the language is reversible, subject

†This work was supported by the Danish Council for Independent Research.

*Corresponding author. E-mail: ups@mmmi.sdu.dk

https://doi.org/10.1017/S0263574710000664 Published online by Cambridge University Press

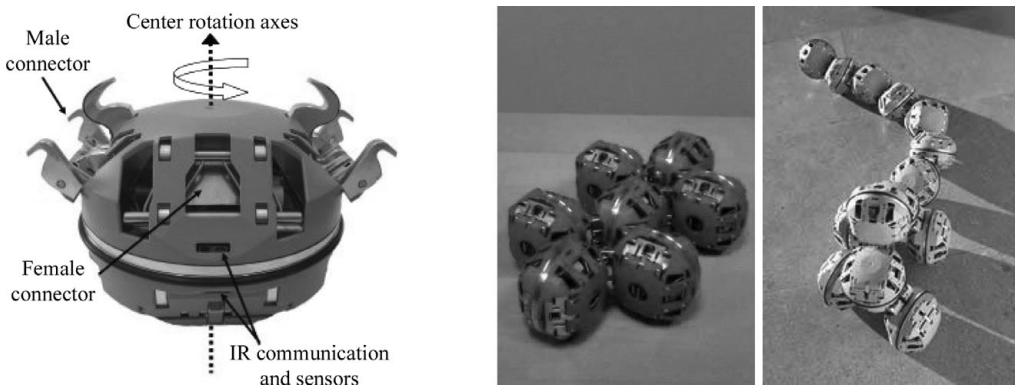


Fig. 1. An ATRON module (left) and various assembled structures (right).

to physical constraints. We demonstrate the effectiveness of our approach in a number of ways: with relatively long-running, reversible self-reconfiguration experiments using the physical ATRON modules, comprehensive self-reconfiguration experiments using the simulated ATRON modules, and a reversible self-reconfiguration experiment using simulated M-TRAN modules.[37]

Reversible self-reconfiguration is useful in many cases because a reverse sequence often is just as useful as the forward sequence. For example, reversibility significantly reduces the amount of work needed to deploy robots in larger scenarios where self-reconfiguration is used to adapt to the environment: as users explore a control strategy while experimenting with the robot in the deployment environment, they can benefit from possibility of automatically reverting to a previous, correctly working configuration. Previous work with the DynaRole language used the DCD virtual machine running on the physical modules as compile target;[7] in this paper we extend this work to allow for compilation to either a TinyOS component, an ANSI C implementation, or a Java implementation (the latter two can be used with our USSR simulation environment).

The rest of this paper is organized as follows. First, Section 2 provides background information on the ATRON robot and discusses related work on self-reconfiguration and reversible programming languages. Then, the main contribution is presented in two parts: first, Section 3 presents our embedded software platform for robust self-reconfiguration followed by Section 4, which presents our high-level language and describes how it compiles to our target platform. The effectiveness of our approach is documented by the experiments in Section 5 and the systematic analysis using simulated modules in Section 6. Last, Section 7 concludes the paper. Our running example throughout the paper is self-reconfiguration of the ATRON robot from a flat "8-like" shape to a car-like shape, both shown in Fig. 1 along with a snake shape. The snake shape is used as a target configuration in specific experiments.

2. Background: Self-Reconfiguration and the ATRON Robot

We now present related work on self-reconfiguration, programming languages for modular robot ensembles, and reversible programming. The related work is followed by background information on the ATRON robot and details on an initial, motivating experiment with the communication capabilities of the physical ATRON modules.

2.1. Self-reconfiguration

Off-line planning of self-reconfigurable robots has been studied for a large number of different robotic systems.[1,2,8,10,22,25,26,30,32,36] These approaches are, however, largely complementary to our work: any off-line planner (for the ATRON or MTRAN robots) could use our scripting language as output and would thus benefit from the robustness and reversibility. Notably, planners are generally based on idealized transition models of the target robots. Our approach complements this work by allowing these idealized self-reconfiguration sequences to be executed more reliably under realistic operating conditions, including changing battery levels, communication problems, and even modules resetting from time to time due to software or electrical failure.

The runtime execution of a self-reconfiguration process on physical robots is treated in some cases; in particular Brandt uses a token-based approach where a single token that is explicitly routed through the module structure is used to represent the active step in the self-reconfiguration sequence.[9] However, this approach is brittle because the loss of the token will cause the system to halt. Replacing Brandt's token-based approach with the diffusion-based approach described later in this paper was experimentally determined to provide a massive advantage in terms of robustness on the physical ATRON modules, which was the initial inspiration for the work presented in this paper. We return to the token-based approach in Section 6, where it serves as a base case for comparison.

On-line, distributed self-reconfiguration algorithms[12,23,27,31,34,37] address the execution issue, which is often ignored in planning. Unlike our system that requires an off-line plan to be computed first, these algorithms allow self-reconfiguration to be done automatically given a target shape. However, in purely on-line distributed algorithms neighbor-to-neighbor communication is essential to determine the position of modules relative to each other and thus a broken communication link, even if it is only one way, is problematic. For example, in the Proteo system by Yim *et al.*, two-way neighbor-to-neighbor communication

https://doi.org/10.1017/S0263574710000664 Published online by Cambridge University Press

is required for coordination between neighboring modules and propagation of heat values in the heat-based method.[34] Lack of two-way communication does not pose a problem to our algorithm because the self-reconfiguration sequence is generated off-line, either by hand or by a planner, and thus we only need to execute actions in the right sequence and not worry about the relative orientation and position of neighbor modules. This implies that our algorithm in theory works as long as there, for any two modules, exists a communication path between them in the robot. The path needs neither to be reliable nor to be static. On the other hand, our algorithm cannot do on-line planning and thus if unexpected problems occur, its only option is to compute an entirely new plan or alternatively run the sequence backwards to restore the initial configuration of the robot. In other words, we gain robustness but loose flexibility.

Robustness is explicitly investigated by Yoshida *et al.* using the Fracta self-reconfigurable robot, but again two-way communication is assumed using a synchronization algorithm, so partial failures in a module would presumably result in the module being rejected from the robot.[21,38] The self-reconfiguration process itself is controlled using a diffusion process, where continuous communication between neighboring modules leads to an equilibrium in the system when the target shape is achieved.[23] Diffusion is also used in our communication protocol, but rather than directly controlling the shape, the diffusion is used to propagate the state of the state machine that drives the self-reconfiguration process.

Robustness is also investigated by Shen *et al.* using a hormone-inspired approach to communication and control in the CONRO self-reconfigurable robot.[28] Hormone signals are packets that are diffused throughout the structure of the robot, possibly causing operations to activate or new hormones to be created when they arrive. Hormones can be used to control locomotion and self-reconfiguration in a highly dynamic fashion that automatically adapts to the current topology. Self-reconfiguration is performed using a "cascade of actions" that is similar to the distributed state machine approach used in this paper. Indeed, our approach can be seen as a simplified version of hormones dedicated to self-reconfiguration; we provide similar properties (dynamic, asynchronous, scalable, and reliable) but, unlike the case for hormones, provide a high-level means of programming the behavior of the system using a distributed scripting language that compiles to the low-level and robust implementation.

2.2. Programming languages for robot ensembles

Zhang *et al.*,[39] De Rosa *et al.*,[17] and Ashley-Rollman *et al.*[3] propose the language-based solutions for the control of modular robot collectives. Similarly to our proposal, their work aims to ease the task of specifying and executing control strategies by giving the programmer a means to target the robot ensemble, rather than each individual robot.

The XML-based scripting language proposed by Zhang *et al.* focuses on locomotion gaits for chain-type modular robots: it allows users to identify functional configurations within the robot ensemble and to assign to them controllers specified as phase automata, so as to achieve the desired group behavior.[39] However, to the best of our knowledge self-reconfiguration is not supported by the language, therefore the issue of temporally constraining the modules' operations and enforcing such constraints at runtime is not explicitly dealt with by the authors.

Locally Distributed Predicates[17] and Meld[3] are two declarative programming languages specifically developed to support the operation of ensembles composed of spherical microrobots.[18] While the context and purpose are similar to the work presented here, a significant difference is the number of robots that ensembles are anticipated to comprise. The spherical microrobots are assumed to exist in numbers several order of magnitudes higher than those made of the ATRON or other macroscale modular robots. Such million-module ensembles can afford to overlook reliability issues that our work needs to address: in cases similar to the one presented in this paper a single failing module can disrupt the whole self-reconfiguration sequence, while in a highly redundant context the same occurrence is not as significant. Nevertheless, our proposal and the declarative-based approach could complement each other in the context of large-scale ensembles made of arbitrary modules, with our solution managing small numbers of tightly coordinated modules forming *metamodule* units.[15] A higher level declarative approach can then be used to drive the reconfiguration at a coarser level.

Although not used for modular robots so far, UrbiScript[4] was among the first languages developed specifically for robotics allowing explicit parallelism control. As this feature was originally introduced to ease the orchestration of complex behaviors for robotic platforms with multiple degrees of freedom (like, e.g., humanoid robots), we found it natural and extremely useful to adapt it to platforms that are similarly capable of many degrees of freedom.

2.3. Programming language reversibility

Our approach to reversible self-reconfiguration is based on a domain-specific language approach where the high-level scripting language is designed such that reversal of arbitrary self-reconfiguration sequences is trivial. General-purpose programs can, however, be made reversible, as demonstrated by Yokoyama *et al.* for the general-purpose, high-level language Janus.[35] Janus allows general algorithms, such as Fast-Fourier Transform, to be implemented and automatically reversed. This approach is much more general than our highly restricted and distributed scripting language, which, for example, does not have a representation of state and control structures. Nevertheless, as demonstrated by Yokoyama *et al.*, the issue of reversing state and control has been solved in general, and the same principles could be used in our language. We note that the work by Yokoyama *et al.* has been highly inspirational for making a reversible language for self-reconfiguration, in particular for details such as designing the operation `rotateFromToBy` (which by design is reversible) as opposed to simply rotating to a specific degree.

2.4. ATRON hardware and software

The ATRON self-reconfigurable modular robot (Fig. 1) is a 3D lattice-type system.[24] Each unit is composed of two hemispheres, which rotate relative to each other, giving the module one degree of freedom. Connection to neighboring

https://doi.org/10.1017/S0263574710000664 Published online by Cambridge University Press

modules is performed by using its four actuated male and four passive female connectors, each positioned at 90 degree intervals on each hemisphere. The likewise positioned eight infrared ports are used to communicate among neighboring modules and to sense distance to nearby objects. Two Atmel ATMega128 micro-controllers (one per hemisphere) linked by an RS-485 serial connection control the hardware. Like most of the other proposed modular robotic systems, the ATRON on-board processing units are severely constrained in order to keep the system simple and potentially cost-effective by mass production: program and data memories are respectively in the order of tens and units of KBytes, while clock is within the ∼20 MHz range and furthermore features common in conventional computers, like a memory management unit (MMU), are absent.

Since the extreme simplicity of the control hardware and its scarce resources prevent system designers from relying on the usual methodologies when engineering the software subsystem, we adopted techniques common in other domains sharing the same constraints, namely sensor networks. The resulting low-level software implementation for the ATRON modules, described in more detail in ref. [5–7], provides the foundations over which we developed the robust approach to self-reconfiguration, described in the next section. We note that this underlying software has already proven to be effective for the development of autonomous controllers in stand-alone, long-running experiments with the physical ATRON modules.[13,29]

2.5. *Communication in the ATRON system*

One of the aims of this work is to make algorithms that can work reliably in a distributed system even if there is unreliable communication between nodes in the system. Unreliability is a fundamental characteristic of most forms of communication, in particular wireless communication because the reliability depends on the relative positions of sender and receiver and is subject to interference from the environment. Moreover, both the relative position and interference will normally change over time.

As a concrete example, let us return to the ATRON self-reconfigurable robot. Each module of this robot has eight transceivers that allow it to communicate with eight neighboring modules. The relative position and orientation of senders and receivers are fixed most of the time, but due to production differences of the transceivers and imprecise mounting the reliability of one communication channel may be very different from another. Furthermore, there is only one micro-controller per four channels and there is a risk of interference between different transceiver pairs. In order to determine the significance of these communication problems we ran 80 experiments where from one to four ATRON modules were programmed to send a message to a central module at intervals of 250 ms. The central module recorded when and where messages were received for later analysis. We did five experiments with one to four senders and for each of these experiments we tested four different configurations. Each experiment was repeated four times. We found that three of the receiver module's communication channels worked equally while one performed significantly worse statistically, requiring up to 20 s for a packet to be transmitted. We also saw a slight increase in communication errors with an increased number of communication channels, but this effect was less significant. The experiments document that in the case of the ATRON robot the assumption of unreliable communication certainly holds in practice despite it should not be the case theoretically. The ATRON communication system could be improved, but it would only reduce the problem, not remove it. In general, these experiments demonstrate that in order to make a reliable algorithm for the ATRON robot and similar systems we need to make algorithms that can handle unreliable communication.

3. Robust Self-Reconfiguration

We define robust self-reconfiguration as a self-reconfiguration process that can tolerate partial hardware failure and still produce the correct target configuration. Concretely, in the context of this paper, we are interested in an approach that tolerates variable running times for operations, partial communication failure, and physical reset of modules; the latter implicitly includes replacing a module with a new one in the same physical state. There are numerous other ways in which modules can fail; we return to this issue later in Section 5. The approach should also allow parallel operations without violating sequential constraints, which is nontrivial given that parallel operations may have variable running times. Note that we are interested in an approach that enables precise control over the self-reconfiguration process for physical modules similar in scale to the ATRON robot, as opposed to, for example, a probabilistic approach that provides robustness through redundancy. Moreover, we are interested in a distributed algorithm, as opposed to a centralized algorithm that has a single point of failure; we do, however, note that the approach we describe here could easily be used with a centralized controller. We now describe the general design of our approach to robust self-reconfiguration followed by details on the communication protocol, how module reset is handled, and how the state machine is implemented.

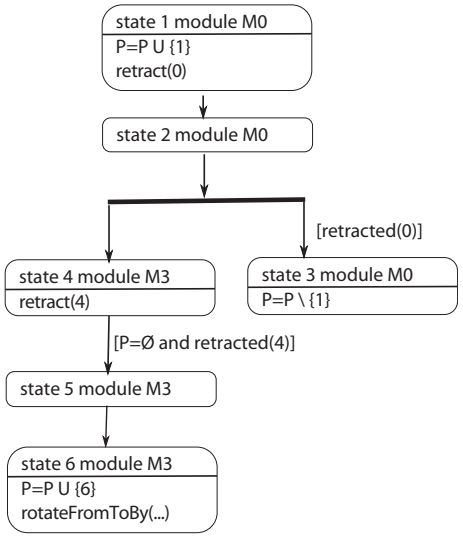
3.1. *General design*

We implement robust self-reconfiguration using a distributed state machine where each module contains a complete implementation of the state machine but only executes those states that are associated with the address of the module. The address is simply an integer that, for example, can be assigned using the preprogrammed internal address of the module or more generally can be assigned based on a number of predicates like we do in this paper (see Section 4). The state machine transfers control between modules by globally sharing the active state and the address of the module that should execute the state; the state sharing is done between neighboring modules using a robust communication protocol based on continuous diffusion of state between neighboring modules, as described later.

Global state is shared continuously between the modules and is used to update the local state of each module. The local state designates the active state currently being executed by the module, if any. Concretely, when the global state received by a module is addressed to the module, the local active state is updated to dispatch to the appropriate action. All

https://doi.org/10.1017/S0263574710000664 Published online by Cambridge University Press

```
switch(state) {
case 1: /* Module M0 */
 call distState.addPendingState(1);
 call Connector.retract[CONNECTOR_0]();
 state = 2;
 break;
case 2:
 call distState.sendState(4,3);
 state = 3; /* fall-through */
case 3:
 if(call Connector.get[CONNECTOR_0]()
    != MALE_CONNECTOR_RETRACTED) break;
 call distState.removePendingState(1);
 state = 255; /* inactive state */
 break;
case 4: /* Module M3 */
 call Connector.retract[CONNECTOR_4]();
 state = 5;
 break;
case 5:
 if(call distState.hasPendingStates()) break;
 if(call Connector.get[CONNECTOR_4]()
    != MALE_CONNECTOR_RETRACTED) break;
 state = 6; /* fall-through */
case 6: /* Module M3 */
 call distState.addPendingState(6);
 rotateFromToBy(0,324,FALSE,150);
 state = 7;
 break;
```



Listing 1. NesC fragment of the distributed state machine for "8 to car" (automatically generated), state diagram shows program flow.

operations are at this level non-blocking and long-running operations run asynchronously to ensure responsiveness of other tasks, such as sharing the global state.

A fragment of the source code of the concrete state machine implementation for the "8 to car" self-reconfiguration is shown in Listing 1 along with a graphical representation of the same states. This fragment of the state machine implements parallel retraction of connectors 0 and 4 on modules M0 and M3, respectively, sequentially followed by a rotation operation on module M3. The variable `state` holds the local active state and is updated from the global active state only when the address corresponds to the appropriate module. API operations are used to maintain the globally shared set of pending states as well as to initiate self-reconfiguration steps and to detect when such steps have completed. Note the use of a `break` after starting a long-running operation to ensure responsiveness. The details of the state machine and its interaction with global state propagation are explained in this section. The state machine could be manually written using the techniques described in this section, but we instead compile it from our high-level language, with evident advantages in terms of speed, convenience, and correctness over hand-made implementations.

3.2. Parallel operations and global synchronization

Given a protocol for globally sharing the state, a distributed state machine that uses a single active state can trivially be used to implement a sequential self-reconfiguration process. To implement parallel operations we use the concept of a *pending state,* which is a state that is still actively executing an operation while the global active state advances. The set of pending states is globally shared and maintained so that when a module starts or completes a pending state this information is propagated to the other modules of the robot. Thus, to implement a sequential operation that executes after all parallel operations have been completed, a given state can wait for the set of pending operations to be empty. In Listing 1, on module M0 the state 1 is added to the set of pending states when the retraction operation is started and removed later (in state 3), when the pending operation is completed. Our approach does not generally support dependencies between independently executing sequences of actions, e.g., execute $S_1; S_2$ sequentially but in parallel with $S_3; S_4$ that also execute sequentially. In practice, this restriction is not a problem for the self-reconfiguration scenarios we are studying as they involve a fairly small number of physical modules; see Section 6 for a discussion of scalability.

The handling of pending states can be seen in the source code fragment of the distributed state machine shown in Listing 1 and in the accompanying state diagram showing the program flow. Here, state 1 on module M0 starts to retract a connector (as a pending operation), adds 1 to the set of pending states (denoted P in the state diagram), and transitions to state 2, which transfers control to state 4 on

https://doi.org/10.1017/S0263574710000664 Published online by Cambridge University Press

module M3 and in parallel continues to state 3, which waits for the connector to be retracted and then removes 1 from the set of pending states. State 4 on module M3 performs a blocking operation: the operation is started but execution of the state machine only continues to state 6 after the operation has completed and there are no more pending states and (this is checked in state 5). Note that each module performs operations such as updating the active state on its local copy of the global state and only synchronizes the local and global state when it transfers control to another module or starts or completes pending states.

3.3. Communication protocol

The global sharing of state is central to the robustness properties of our approach to self-reconfiguration. First, global sharing of state circumvents partial communication failure, because if there is a communication path between two modules, information will eventually propagate between the two modules. Second, global sharing of state helps to tolerate reset of individual modules since the state of the individual module can be restored from the neighbors; a more detailed analysis of tolerating module reset can be found below. All communication between modules is performed using idempotent messages, meaning that they can simply be transmitted repeatedly throughout the self-reconfiguration process, which increases tolerance toward unstable communication where only a small percentage of messages gets through. We refer to this approach as *state diffusion*, since any local state advancement is eventually propagated throughout the structure until it reaches an equilibrium (all modules having the same state). In addition to robustness, the protocol is oblivious to network topology, asynchronous, and scalability.* The protocol is oblivious to network topology in the sense that state will be propagated so long as there, at some point in time, is at least a one-way connection between the parts of the robot. The protocol is asynchronous since advancements to the active state and pending states can propagate independently throughout the structure while operations are being performed. The protocol is scalable because the number of packets sent per second per active connector is constant, the specific number of packets can even be tuned to the physical constraints of the system, which enables developer to balance fast state propagation and computational load due to communication.

Our communication protocol is designed to share the active state, the address of the active module, and the set of pending states using idempotent messages. The set of pending states grows and shrinks as pending operations are added and completed. Each module continuously diffuses (i.e., locally broadcasts) packets that contain the local copy of the global state and is responsible for merging copies of the global state received over the network. Updates are always made to the local copy: after completing an operation a module can update the local copy of the active state and active module address (according to the state machine transition) and update the local set of pending states by adding or removing elements. An update is propagated throughout the module structure by the continuous transmission of local state to neighboring modules that in turn merge their local state with the incoming updated state. The merge function is shown in Fig. 2. The two key properties that we exploit are (1) that a pending state p_0 added to an active module M_0 in active state s_0 is *always* added before the active state is propagated to some other module M_1, and (2) the active state s_1 propagated to the other module M_1 is greater than p_0. This implies that when merging "older" incoming global states, which have a lower active state, removal of pending states should only be taken into account for those pending states that the incoming global state could have known about, that is, those pending states that are lower than the active state of the incoming global state. The inverse relation holds for merging "newer" incoming global states, which have a higher active state. A key property of this merge function is that removal of pending states can propagate along the same path as the active state is being transferred, improving tolerance toward partial failure of communication in comparison to, for example, an algorithm based on first reaching consensus over the global active state. Such an algorithm would require the newer state to be propagated back for the resolved pending states to be appended and then taken into account.

$$\left.\begin{array}{ll}\text{Local state} & G_0 = (s_0, a_0, P_0) \\ \text{Incoming state} & G_1 = (s_1, a_1, P_1)\end{array}\right\} \text{Resulting state } G_2$$

$$G_2 = \begin{cases} (s_0, a_0, \{p \in P_0 | p > s_1 \vee p \in P_1\}, & s_0 > s_1 \\ (s_0, a_0, P_0 \cap P_1), & s_0 = s_1 \\ (s_1, a_1, \{p \in P_1 | p > s_0 \vee p \in P_0\}, & s_0 < s_1 \end{cases}$$

Fig. 2. Global state merge function. Let $G_i = (s_i, a_i, P_i)$ denote a copy of the global state where s_i is the currently active state, a_i is the address of the active module, and P_i is the set of pending states. Let G_0 be the local copy of the global state on a module, G_1 be the incoming global state being received over the network, and G_2 be the resulting global state, which will be propagated to all neighbors of the module.

The communication scheme is exemplified in Fig. 3 for the first few steps of the "8 to car" self-reconfiguration of Listing 1. The corresponding state transitions for modules M0 and M3 are shown in Table I. Module M0 starts a pending operation and advances the global active state to 4, while maintaining the local active state 3 to wait for the pending operation to complete (steps 1–3). In step 4 the global active state is propagated but fails to propagate directly to M2, although it eventually propagates to M3 and M2 (steps 4–6). At this point module M3 waits in active state 4 until the local retract operation completes and the set of pending states becomes empty. The pending operation on M0 completes (step 6) and the updated pending set is propagated to the rest of the modules (steps 7 and 8). Note that due to the design of the merge function (Fig. 2), the pending set information is propagated from M0 to the rest of the structure even though the global active state has the same value. Eventually, the retract operation on M3 has completed and the set of pending states

* These properties are due to Shen *et al.*;[28] in addition, we note directed diffusion in sensor networks as a source of inspiration[19] for the diffusion concept.

https://doi.org/10.1017/S0263574710000664 Published online by Cambridge University Press

Table I. Global state sharing for the "8 to car" example, a graphical overview of the corresponding steps is shown in Fig. 3. The local active state (denoted with superscript) corresponds to the actions of the state machine in Listing 1. The local active state is advanced when a module performs actions and is propagated through the global active state (first tuple element) to the module that is to perform the next action (identified by the second tuple element). The set of pending states (the third tuple element) tracks the start and stop of operations happening in parallel.

	⟨state, address, pending⟩$^{\text{active}}$		
Step	M0	M3	
(1)	$\langle 1, M0, \{\}\rangle^{1}$	$\langle 1, M0, \{\}\rangle^{-}$	initial state
(2)	$\langle 2, M0, \{1\}\rangle^{2}$	$\langle 1, M0, \{\}\rangle^{-}$	start retracting connector 0
(3)	$\langle 4, M3, \{1\}\rangle^{3}$	$\langle 1, M0, \{\}\rangle^{-}$	advance global active state, local active state waits for pending operation
(4)	$\langle 4, M3, \{1\}\rangle^{3}$	$\langle 1, M0, \{\}\rangle^{-}$	global active state propagation with partial error
(5)	$\langle 4, M3, \{1\}\rangle^{3}$	$\langle 4, M3, \{1\}\rangle^{4}$	global active state reaches target, start operation
(6)	$\langle 4, M3, \{\}\rangle^{-}$	$\langle 4, M3, \{1\}\rangle^{4}$	M0 pending operation completes, remove pending state
(7)	$\langle 4, M3, \{\}\rangle^{-}$	$\langle 4, M3, \{1\}\rangle^{4}$	global active state propagation, M3 waits for pending state completion
(8)	$\langle 4, M3, \{\}\rangle^{-}$	$\langle 4, M3, \{\}\rangle^{4}$	global active state propagation
(9)	$\langle 4, M3, \{\}\rangle^{-}$	$\langle 5, M3, \{\}\rangle^{5}$	advance global and local active states

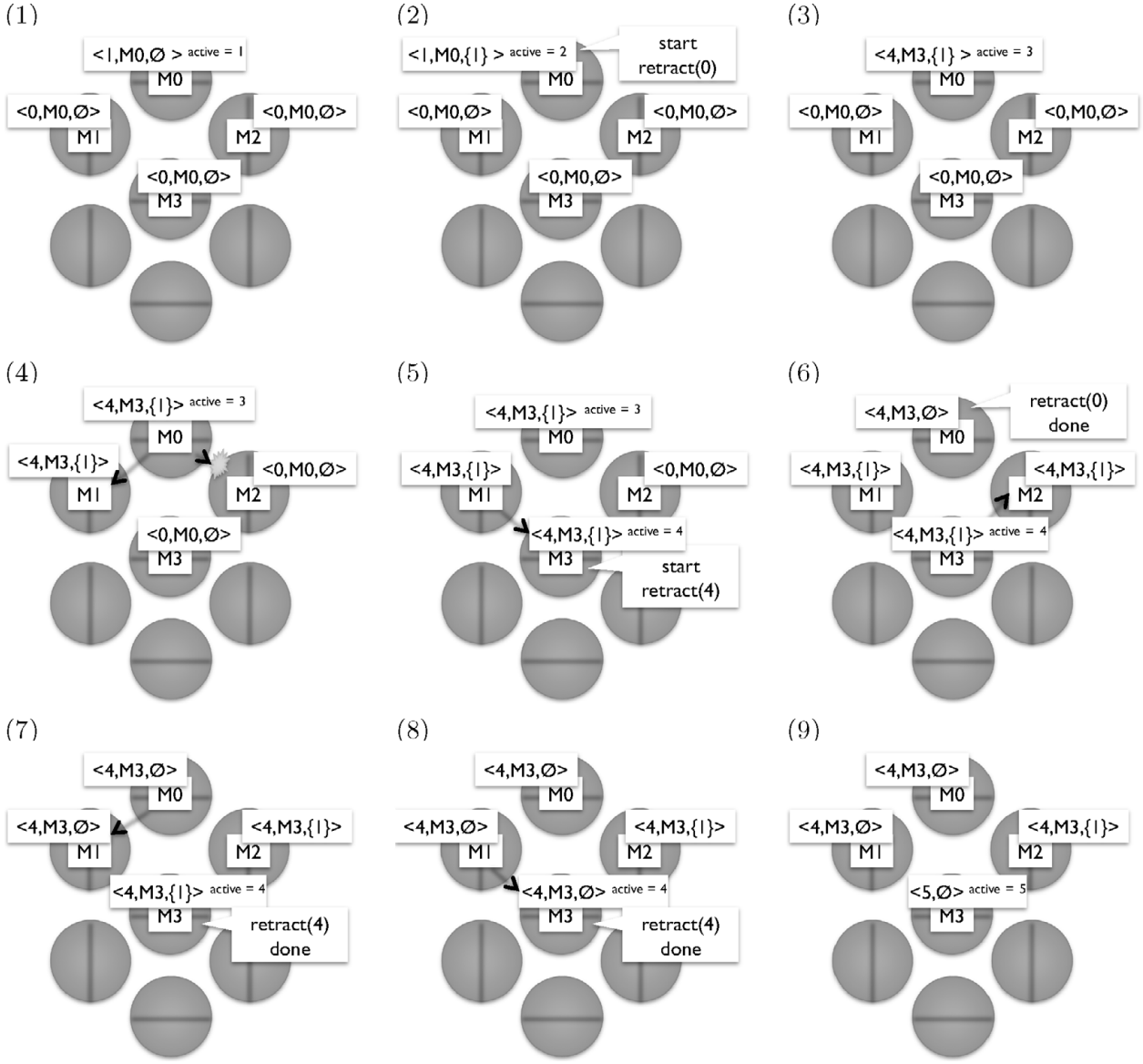


Fig. 3. Overview of global state sharing example for the "8 to car" example, illustrates state transitions shown in Table I for the initial "eight" configuration (lines are main joint rotation axes). The local active state (denoted with superscript) corresponds to the actions of the state machine in Listing 1. The local active state is advanced when a module performs actions and propagates through the global active state (first tuple element) to the module that is to perform the next action (identified by the second tuple element). The set of pending states (the third tuple element) tracks the start and stop of operations happening in parallel.

https://doi.org/10.1017/S0263574710000664 Published online by Cambridge University Press

is empty, allowing M3 to advance to active state 5 (step 9). Note that state propagation is asynchronous and could happen differently with respect to when operations complete and when state is propagated to other modules, but that in all cases the two retract operation on modules M0 and M3 will complete before the global active state advances to 5. The global state sharing could be done in numerous other ways; we consider the evaluation of alternative approaches to be future work.

3.4. Module reset

Numerous kinds of hardware and software faults can occur during a self-reconfiguration sequence; we are concerned with a specific fault, namely spurious reset of a module either due to hardware faults or induced by e.g. a watchdog timer firing because of a software error. Using our shared state approach, a module that is not currently performing an operation can trivially tolerate a reset in the middle of a self-reconfiguration sequence. The state will be restored from the neighbors, and if the module was to perform the next action, the global state will simply not be advanced until the module is ready and starts to execute this state. A more critical case is reset of a module that is in the middle of performing an operation. Such a module can in many cases be reset, but only when all API operations are idempotent, which is the case for, e.g., ATRON. Specifically, we use idempotent operations, such as "extend connector" or "rotate to position 324," and such operations will under normal circumstances simply complete when power is restored and the global state is propagated to the module.

Reset of a module that is performing a pending operation requires special support: the state of the module will, after restarting, be reset to the global state, which is higher than the pending state. Each module locally keeps track of the pending states that it has started; this information will be erased after a reset, which can be used to re-enter the pending state when required. By combining the local set of pending states that have been started with the information of what states are globally pending and what pending states in a given module are responsible for completing, a module can detect the situation where it is responsible for a pending state but has not removed it from the global state because it was stopped in the middle of the operation. Simply re-entering the pending state will normally cause the pending operation to complete and hence provide the desired robustness. Specifically, in simulation such pending operations always complete, whereas on the physical modules spurious errors have been observed in some cases when restarting an operation that actuates the main joint.

A critical issue is that the first action of the first module of the self-reconfiguration sequence must be able to detect if a new self-reconfiguration sequence is being started or if the module has just been reset in the middle of an ongoing sequence. In the presence of unreliable communication, the module cannot safely determine whether to initiate the first step of the sequence or wait to have its state restored from the neighboring modules. For this reason we simply use a timeout after which the first module proceeds with the self-reconfiguration sequence.

3.5. Implementation

Due to the necessarily simple design of the ATRON hardware, communication among neighboring modules can be heavily constrained and its performance can drop significantly, as the four communication channels on each hemisphere rely on a single multiplexed UART and that, furthermore, given the half duplex nature of the infrared channel, packet collisions can easily happen. While we are developing a new version of the ATRON electronics to reduce the significance of this issue,[11] we currently rely on compile-time virtualization of the hardware[20] and address the low communication performance using the diffusion approach. Communication is implemented as local continuous broadcast with a simple random desynchronization. As outlined, the communication protocol is explicitly designed to support this form of redundant communication as the packets are idempotent and thus multiple receptions of the same message do not halt or disrupt in other ways the distributed state machine implementing the self-reconfiguration process.

Concretely, the implementation is structured as two components: a state machine, which is specific to the self-reconfiguration sequence, and a state sharing component that maintains the shared global state, including the pending states. When automatically generating the state machine implementation, as is done in the next section, only the state machine component needs to be generated. As a side note, we mention that having precise control on whether an operation should block or execute in parallel but with a signal upon termination significantly simplifies the implementation of blocking and pending operations: our TinyOS-based implementation fully supports this semantics,[6] and we are working on the integration of similar features in our virtual machine-based runtime implementation.[7] As both implementations run on the extremely resource-constrained computational infrastructure of the current ATRON revision, our approach constitutes a general solution suitable for a wide class of self-reconfigurable platforms.

4. Programming Reversible Self-Reconfiguration Sequences

The distributed state machine design described in the previous section provides robust self-reconfiguration with safe parallel operations. Manual implementation of such a state machine is however tedious and error-prone: the association between modules and states must be maintained, blocking operations must be inserted where needed but without rendering the state-sharing unresponsive, and pending states must be handled correctly. The state machine implementation could be automatically generated from a planner, but each implementation of a planner would then need a carefully engineered code generator to correctly generate code matching the state machine design. Moreover, manual optimizations, such as parallelization, would require the programmer to manually modify the state machine implementation, which could lead to programming errors. To resolve these issues, we have implemented a distributed scripting language that can be directly used by the programmer to quickly and concisely specify self-reconfiguration sequences or alternatively can

https://doi.org/10.1017/S0263574710000664 Published online by Cambridge University Press

Table II. The DynaRole ATRON API, selected operations.

`M.Connector[`*n*`].retract()`
retracts connector number *n* (releasing a connected module, if any).

`M.Connector[`*n*`].extend()`
extends connector number *n* (connecting to an appropriately positioned module, if any).

`M.rotateFromToBy(`*f*`,`*t*`,`*d*`,`*s*`)`
rotates the main joint from *f* degrees to *t* degrees in direction *d* at speed *s* (behavior is undefined if the joint is not at *f* when starting).

be used as code generation target by a planner. Furthermore, due to the high-level nature of the language, reverse self-reconfiguration sequences can be automatically generated from the usual forward ones. We now describe this language in more detail, first the syntax and semantics, then the compilation process, and last the reversibility and generality.

4.1. Syntax and semantics

We have designed and implemented our distributed scripting language as an extension to the DynaRole language.[7] In the DynaRole language, roles are used to encapsulate sets of behaviors that should be activated on specific modules in a structure. The assignment of roles is declarative and is used as a basis for dynamically updating behaviors in a running system using a virtual machine approach. Self-reconfiguration, however, concerns multiple tightly coordinated modules performing a number of operations, which is difficult to encapsulate using the concept of a role. As an alternative, we have implemented a new construct, the *sequence*, which is a number of operations that are executed across a number of modules. The concept of a role is still used to identify which modules perform what operation, although that topic is not investigated in detail in this paper.

As a concrete example, consider the sequence shown in Listing 2, which describes the complete "8 to car" self-reconfiguration process. Each statement is prefixed with a label indicating what module is executing the statement, e.g., `M0` is used to indicate a specific module. Following the label is a call to the ATRON API, for example controlling the connectors or the main joint; see Table II for details. Note that the rotation call has been augmented with an extra argument to facilitate reversing the program, as described later.

Each statement is terminated either with a semicolon ";" meaning sequential execution (the next statement is dependent on this operation) or an ampersand "`&`" meaning parallel execution (the next statement is independent of this operation), similarly to, e.g., UrbiScript.[4] A sequence of parallel statements are considered independent, i.e., physically unconstrained, and may be executed in any order but must be completed before the next sequential execution point. As an example, consider the first four lines of the "8 to car" sequence, which indicate that modules `M0` and `M3` can open their connectors in parallel, whereas the rotation of modules `M3` and `M4` must be done sequentially and must only take place after both connectors have opened. We note that nesting of sequential statements inside parallel statements across multiple modules is not currently supported (as has been described earlier, this feature is not supported by the current state machine design).

```
sequence eight2car {
  M0.Connector[$CONNECTOR_0].retract() &
  M3.Connector[$CONNECTOR_4].retract();
  M3.rotateFromToBy(0,324,false,150) ;
  M4.rotateFromToBy(0,108,true,150);
  M4.Connector[$CONNECTOR_0].extend() &
  M1.rotateFromToBy(0,324,false,150) &
  M6.Connector[$CONNECTOR_2].retract();
  M4.rotateFromToBy(108,216,true,150) &
  M6.rotateFromToBy(0,108,true,150);
  M0.Connector[$CONNECTOR_0].extend();
  M6.Connector[$CONNECTOR_6].retract();
  M0.rotateFromToBy(0,324,false,150) &
  M1.rotateFromToBy(324,0,true,150);
  M0.rotateFromToBy(324,0,true,150);
  M5.Connector[$CONNECTOR_0].extend() &
  M2.Connector[$CONNECTOR_4].extend() &
  M1.Connector[$CONNECTOR_4].extend();
  M4.Connector[$CONNECTOR_0].retract();
  M3.Connector[$CONNECTOR_6].retract();
  M1.rotateFromToBy(0,108,true,150) &
  M3.rotateFromToBy(324,0,true,150);
  M1.Connector[$CONNECTOR_6].extend();
  M3.Connector[$CONNECTOR_0].retract() &
  M3.Connector[$CONNECTOR_2].retract();
  M1.rotateFromToBy(108,216,true,150);
}
```

Listing 2. DynaRole sequence describing the "8 to car" self-reconfiguration sequence.

The labels that indicate what module should execute a given statement are defined using roles, which again are defined using logical predicates on the local state and the context of each module. (The context is defined as the state of the immediate neighbors.) In this paper, we, for simplicity, only use a local predicate on the internal ID of the module, which is programmed when flashing the module. For concrete examples of more general predicates, see Bordignon *et al* .[7]

4.2. Compilation process

We have implemented a backend for the DynaRole compiler that can generate either a nesC implementation that executes in the TinyOS-based environment described in Section 2.4 or alternatively an ANSI C or Java implementation that can execute in the USSR simulation environment for modular robots.[14] Concretely, the compiler generates an implementation of the distributed state machine described in Section 3 and assigns the module address based on evaluating the role requirement predicates. The generic parts of the state machine (state management, communication, etc.) are generated using a simple template-based code generator. The sequence implementation is used to generate the body of

https://doi.org/10.1017/S0263574710000664 Published online by Cambridge University Press

the state machine, one statement at a time. For each statement, a number of states are generated, depending on whether the statement is parallel or sequential.

- For a parallel statement a state S_1 is generated that adds S_1 to the set of pending states, executes the operation(s) designated by the statement, updates the global state to the state of the next statement S_3, and then locally transitions to state S_2. The state S_2 monitors the pending operation and updates the global state to delete it when the operation completes.
- For a sequential statement (including the last of a sequence of parallel statements) a state S_1 is generated that executes the operation(s) designated by the statement and then transitions to state S_2. The state S_2 waits for all pending states to be deleted and for the operations of the statement to be terminated after which it transitions to the state of the next statement S_3.

The transition between two statements that are on the same physical module is, when possible, simply performed directly without updating the global state.

4.3. Reversibility

Given a distributed sequence written in DynaRole, it is straightforward to generate the reverse sequence: the ordering of statements must be reversed while retaining dependence relations between statements, and each statement must in itself be reversed. Reversal of the ordering of the statements must preserve the semantics that a sequential dependence between two statements requires all parallel statements to have completed before the next sequential statement can execute. This implies that the statements can be ordered in reverse while retaining the same separator between each given pair of statements. For example, a sequence of statements $S_1 \& S_2; S_3;$ reverses to $S_3; S_2 \& S_1;$. Reversal of an operation is straightforward due to the design of the API operations supported in DynaRole sequences: `retract` becomes `extend` and *vice versa*, whereas `rotateFromToBy` simply swaps the from and to angles and reverses direction; see related work in Section 2 for a discussion of reversal of non-API operations.

The programmer explicitly defines and invokes reversed sequences, for example, the sequence shown in Listing 2 is reversed using the following declaration:

```
sequence car2eight = reverse eight2car;
```

After this declaration the name `car2eight` can be used like any other sequence name.

4.4. Generality

We believe that DynaRole could be used to program reversible self-reconfiguration sequences for most if not all modular self-reconfigurable robot systems, although not all sequences will necessarily be reversible. Specifically, given a self-reconfigurable robot that can change from configuration C_0 to C_n through a number of intermediate configurations C_i and where each step from C_i to C_{i+1} can be reversed bringing the robot back to configuration C_i, the entire self-reconfiguration sequence is reversible using DynaRole. Not all operations can be reversed on all robots, for example, due to disconnection of the structure, gravity, or a change in the environment; similarly a connector mechanism that requires different physical movements to connect and disconnect might not be reversible due to motion constraints. Thus, programming reversible self-reconfiguration sequences require the programmer to use only reversible steps and moreover might require the design of a high-level API that lets the programmer work in terms of abstract macro-steps (e.g., "connect" and "disconnect" operations for a connector mechanism requiring different physical movements).

The program reversal performed by the DynaRole compiler works by reversing each API call. For this reason, the compiler provides a simple plugin model allowing reversal of different APIs to be implemented. Concretely, the DynaRole compiler currently supports ATRON and MTRAN APIs.

5. Experiments

We now present the demonstration experiments performed to validate the claims made in this paper. Three experiments are performed with the physical ATRON modules and one experiment is performed with simulated MTRAN modules.

5.1. Physical ATRON modules

The first three experiments are performed using the physical ATRON modules running nesC code generated by our compiler from scripts written in DynaRole. The first experiment serves as an overall documentation that our compiler works, generates reverse sequences, and enables a long-running self-reconfiguration process. The second experiment provides a verification of selected robustness claims. The third experiment puts the self-reconfiguration sequence that we considered so far in the paper, the so called "8 to car," in the context of a more general reconfiguration process, showing the generality of our approach.

The first experiment demonstrates the "8 to car" self-reconfiguration sequence implemented by the program of Listing 2. The forwards self-reconfiguration sequence runs first followed by the reverse sequence, after which the process is automatically restarted. Over three experiments the complete forwards–backwards self-reconfiguration sequence completes every time (no restarting) and with restarting it ran the complete forwards–backwards sequence three times, after which a hardware failure in a module caused the process to terminate. This experiment thus demonstrates that our compiler can generate not only correct forwards-executing code but also correct backwards-executing code, in both cases correctly handling dependencies between operations. Moreover, given that the ATRON modules are prone to hardware-induced communication failures, the fact that the "8 to car" sequence completely demonstrates the robustness of our approach.

The second experiment investigates the robustness claims: during the "8 to car" sequence we power off a module before it receives the active state (Fig. 5) and power off a module while it is performing a connection operation (Fig. 6, left). In both cases, the self-reconfiguration process continues

https://doi.org/10.1017/S0263574710000664 Published online by Cambridge University Press

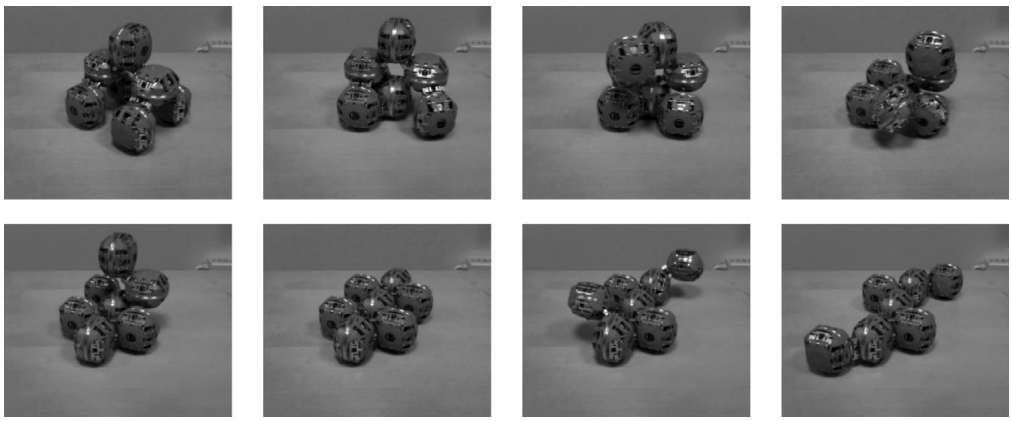


Fig. 4. Self-reconfiguration from car to snake; the self-reconfiguration process subsequently reverses and returns the robot to the car shape (not shown).



(a) (b) (c) (d) (e)

Fig. 5. Robustness toward inactive module begin reset: (a) initial configuration; (b) manually powering off module; (c) sequence blocks, waiting for inactive module; (d) module powered on again; (e) sequence continues.

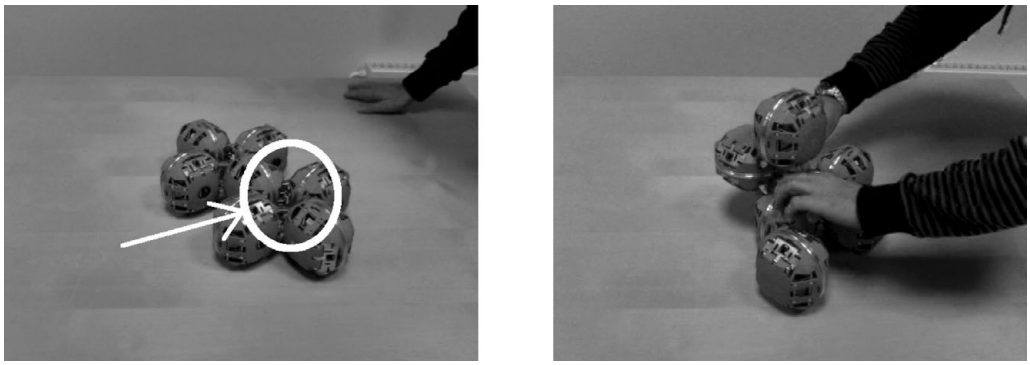


Fig. 6. Left: Robustness toward an active module being reset, the highlighted module is reset while opening a connector. Right: Robustness toward failing communication demonstrated by manually blocking communication paths in turn.

when the module is powered on again. Moreover, we also physically obstruct the communication paths one at a time between the module starting the self-reconfiguration process and the module taking the subsequent step, and in all cases the self-reconfiguration continues (Fig. 6, right). Thus, we have demonstrated a tolerance toward certain kinds of partial hardware failures in modules.

Last, the third experiment extends the "8 to car" sequence used as a running example throughout the paper by first performing the reverse "car to 8" transformation and then chaining a further transformation from the "8" shape to a snake-like configuration, then back again to the original car. The first half of the self-reconfiguration sequence is shown in Fig. 4. This last example of a reconfiguration between two ATRON morphologies that are often used for locomotion is illustrative of the general usefulness of an expressive, efficient, and reliable means for self-reconfiguration in a more general context.

5.2. Hardware failures

The ATRON modules can in some cases spuriously reset due to unknown errors (hardware or software), but are in such cases more likely to lock up, thus blocking the self-reconfiguration sequence. Spurious resets may have occurred during the experiments, but due to the state-sharing properties of our approach would not have been detected. Manually cycling the power on a module would in some cases cause blocked self-reconfiguration sequences to continue; we believe that a watchdog timer triggering a reset could in many cases be used to automatically achieve a similar effect. On a few occasions a module would perform an incorrect action such as not rotating to the correct target position or opening a connector, thus detaching a module from the structure. This last category of errors is out of scope of our approach and will always cause the self-reconfiguration sequence to fail.

We did not attempt to detect communication errors on the physical modules, because messages in most cases

https://doi.org/10.1017/S0263574710000664 Published online by Cambridge University Press

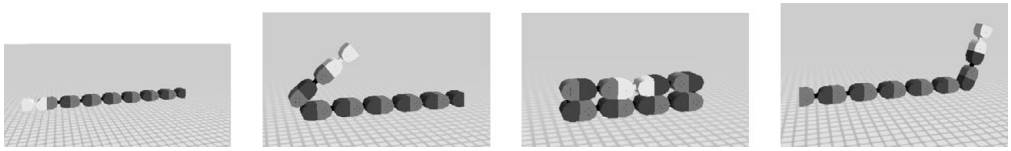


Fig. 7. Reversible reconfiguration with simulated MTRAN modules: displacing a module from the left to the right is manually programmed, whereas the reverse self-reconfiguration is automatically derived.

would eventually be propagated between modules due to the properties of the diffusion protocol. A notable exception is that some modules had connectors that were completely unable to communicate, which could cause the self-reconfiguration sequence to stop in specific configurations where the only communication path between parts of the structure was through faulty connectors.

5.3. Simulated MTRAN modules

The last experiment is performed using simulated MTRAN modules, and serves to illustrate the generality of our approach. The simulation is performed using USSR, a generic simulator supporting several different modular robots, including the ATRON and MTRAN robots.[14] The experiment is a simple reconfiguration of a snake where a module is moved from one end of the robot to the other, see Fig. 7; as was the case for the ATRON robot the reverse sequence is automatically derived. Support for MTRAN in DynaRole simply required the implementation of the statement reverser for the MTRAN API (which for our purposes is largely equivalent to the ATRON API except that rotation includes an extra argument indicating what actuator to use).

6. Analysis

The diffusion-based communication algorithm presented in the previous sections provides a significant improvement in robustness over previous approaches used on the physical ATRON modules. This improvement might however be due to specific characteristics of the ATRON hardware and low-level software. To clarify this issue, we analyze the effectiveness of the diffusion-based communication scheme, compared to a token-based version based on the algorithm described by Brandt.[9] Using the USSR simulator for modular robots, we simulate various degrees of unreliability in the underlying hardware, and measure the resulting performance of the algorithms. We first explain the experimental methodology, then present a number of simulated experimental results, and last present an overall assessment.

6.1. Methodology

To analyze the effectiveness of the diffusion-based algorithm on different types of hardware, we run simulated experiments with the "8-to-car" and "8-to-snake" self-reconfiguration sequences as well as a specific snake configuration used to investigate scalability. All experiments are done using the USSR simulator for modular robots, time is reported using simulated time (and hence is, for all practical purposes, oblivious to the machine running the simulation), and experiments are repeated for 20 times. Each communication channel on a module is restricted to only transmit packets every 100 ms (a worst-case assumption compared to the physical ATRON modules). Average running times are computed using only those experiments that completed the sequence, which leads to a number of ceiling and floor effects[16] that we discuss as the results are presented.

Table III. Communication algorithms included in our analysis.

Diffusion sequential	Diffusion-based, sequential
Diffusion parallel	Diffusion-based, parallel
Token 32 retries	Token-based (sequential), max 32 retries
Token 128 retries	Token-based (sequential), max 128 retries
Token unlimited retries	Token-based (sequential), unlimited retries

The algorithms included in our study are listed in Table III. The two variants of the diffusion algorithm are automatically generated from corresponding DynaRole programs (e.g., the program in Listing 2 for the "8-to-car" sequence) using the Java backend of the compiler. The sequential version was generated by first replacing all parallel statements with sequential statements, whereas the parallel version was generated directly. The token-based algorithm based on the description by self-reconfiguration sequence also computes the shortest path from one active module to the next and explicitly routes the state changes from module to module. The "8-to-car" sequence is implemented based on the output of the planner, whereas the "8-to-snake" sequence was implemented manually using the same principles. The original algorithm, however, assumes reliable communication, which makes it irrelevant as a basis of comparison for experiments investigating fault-tolerance. For this reason, we have included three versions of the algorithm that differ based on how many times retransmission of a packet is performed. The higher the number of retransmissions the higher the reliability but conversely the greater the risk that bandwidth is wasted on retransmitting packets that have already arrived. (An implementation could use an exponential backoff scheme that would approximate the optimal performance of these three variants.) All token-based algorithms perform self-reconfiguration sequentially.

The diffusion-based and the token-based algorithms are fully distributed algorithms in the sense that there is no central controller. The analysis presented in this section however also applies to centrally controlled self-reconfiguration: a centralized controller would still need to communicate with the modules in the structure, and this could be either diffusion-based or token-based communication. Compared to the distributed control, a centralized controller would in most cases have longer communication pathways making it more sensitive to, for example, packet loss, but the overall behavior would be similar for the scenarios we investigate in this section.

https://doi.org/10.1017/S0263574710000664 Published online by Cambridge University Press

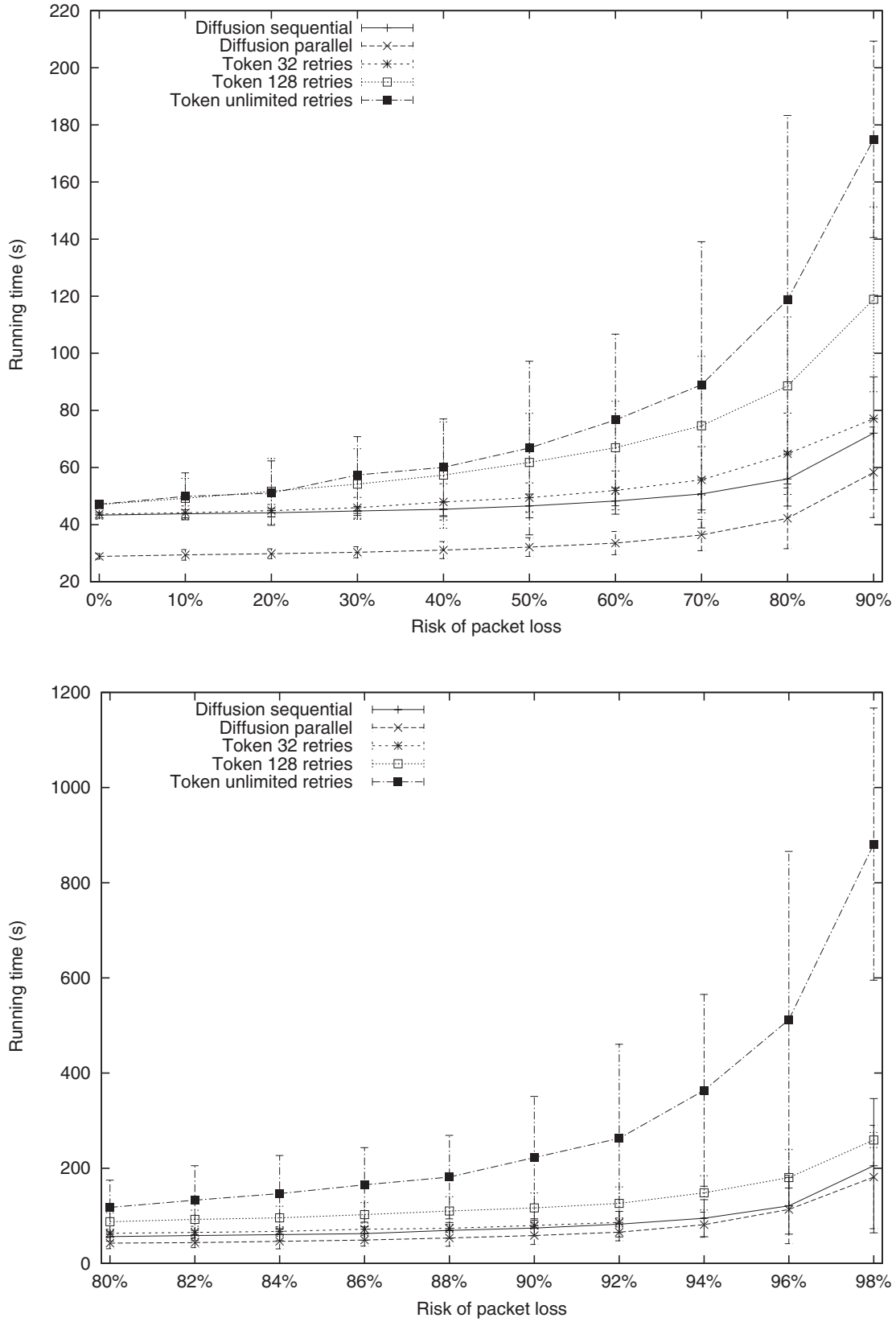


Fig. 8. Time for self-reconfiguration at different rates of packet loss; upper graph shows overview of all experiments, and bottom graph shows details of selected experiments.

6.2. Efficiency versus packet loss

The first simulated experiment measures the effect of packet loss. We vary packet loss from 0 to 98% and measure execution time, success rate, and the number of packets transmitted. The execution time at different rates of packet loss is illustrated in Fig. 8, the bottom graph providing a detailed view of the execution time between 80% and 98% packet loss. The parallel algorithm is the fastest, the token-based unlimited retries is the slowest, whereas the other algorithms are in-between. The execution times of the

https://doi.org/10.1017/S0263574710000664 Published online by Cambridge University Press

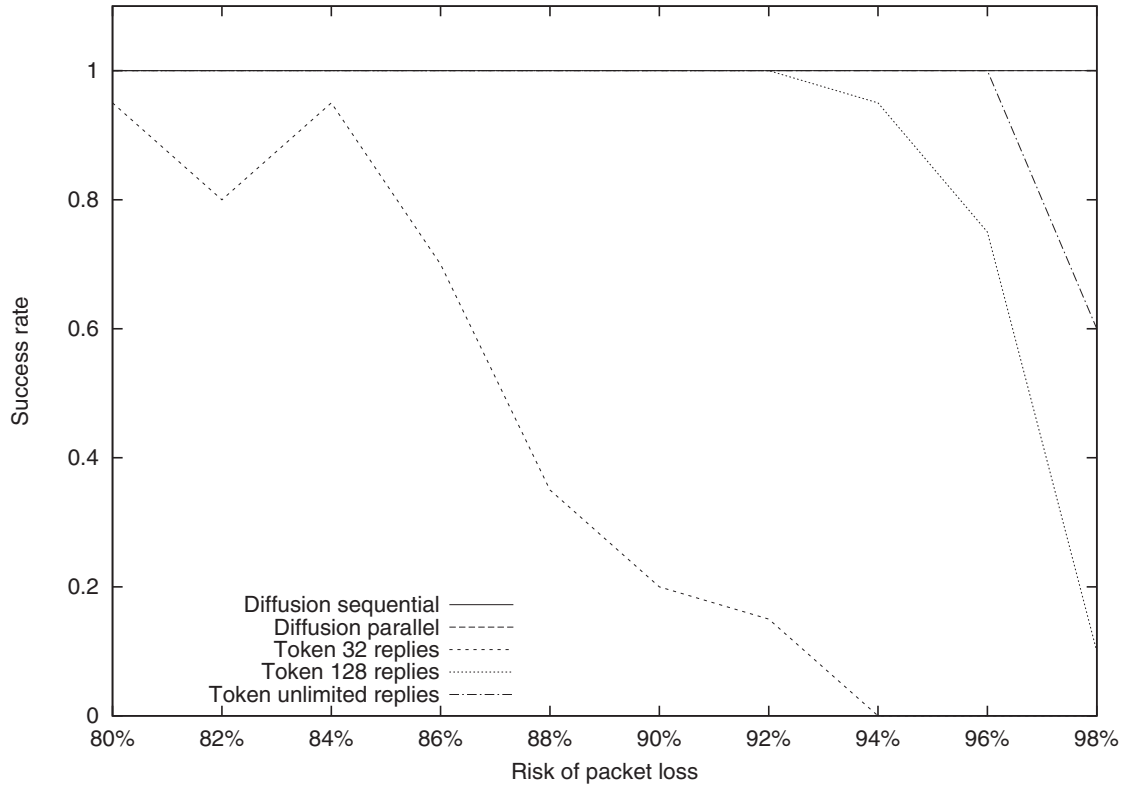


Fig. 9. Robustness of self-reconfiguration at different rates of packet loss, corresponding to Fig. 8.

token-based algorithms are, however, distorted due to a ceiling effect: the self-reconfiguration runs that did not complete are not included in the averages. The success rates of various algorithms for high rates of packet loss are shown in Fig. 9: above 84% token-based 32 retries becomes very unreliable, whereas token-based 128 retries tolerates higher levels of packet loss but ultimately becomes unreliable as well. In both cases the self-reconfiguration will stop at some point because of the token essentially becoming "lost." The token-based unlimited retries algorithm cannot loose the token, but exceeds 1000 s simulated time (the maximal set for this experiment) in a number of cases.

Packet counts for the self-reconfiguration experiments are shown at different rates of packet loss in Fig. 10. The token-based algorithms use approximately the same number of packets at low rates of packet loss; the 32-retries algorithm remains at a fairly constant rate but is omitted at 90% because of the floor effect of the self-reconfiguration sequence stopping prematurely. The constant rate is due to the number of packets always being transmitted before the receiving module has sent a confirmation. The unlimited retries shows an increase in the number of packets corresponding to increase in execution time, and eventually surpasses the diffusion-based algorithms, which transmit much larger number of packets due to the constant use of local broadcast. Packet counts are investigated later in Section 6.7 in terms of scalability.

6.3. Robustness versus faulty connectors

A physical ATRON module tends to have a high degree of variability in the reliability of communication across its different connectors. In addition to typical electronic hardware faults, this can happen because of slight placement offsets of the infrared diodes during the module manufacturing process. Furthermore, the position and the physical integrity of the diodes can also be compromised by operating the connectors when two modules are too close: the mechanical latch can "grip" on the diodes instead of its female receptacle, misplacing or damaging them. In some cases, modules are even completely unable to communicate on specific connectors. In order to investigate this issue in more detail, we measure the success rate of the "8-to-car" self-reconfiguration sequence with varying degrees of connector defects. For each rate of defect communication connectors risk we generate a fixed set of broken connector experiments that are reused between algorithms, thus ensuring that the 20 experiments for each algorithm are run on the same set of module defects for a given risk of connector defects. The results are shown in Fig. 11. The x-axis denotes the risk of each individual connector being defect in a given experiment (there are $7 \times 8 = 56$ connectors in total in the robot). The two diffusion-based algorithms exhibit identical performance, which is superior to that of the three token-based algorithms that also exhibit identical performance. The results basically demonstrate the sensitivity of the fixed routing used by the token algorithm, whereas the diffusion algorithms are more robust because they propagate state throughout the module structure if there is a path.

6.4. Overall performance

As an alternative to complete failure of connectors, we can more interestingly model a heterogeneous set of connectors where there is a general packet loss and a set of connectors

https://doi.org/10.1017/S0263574710000664 Published online by Cambridge University Press

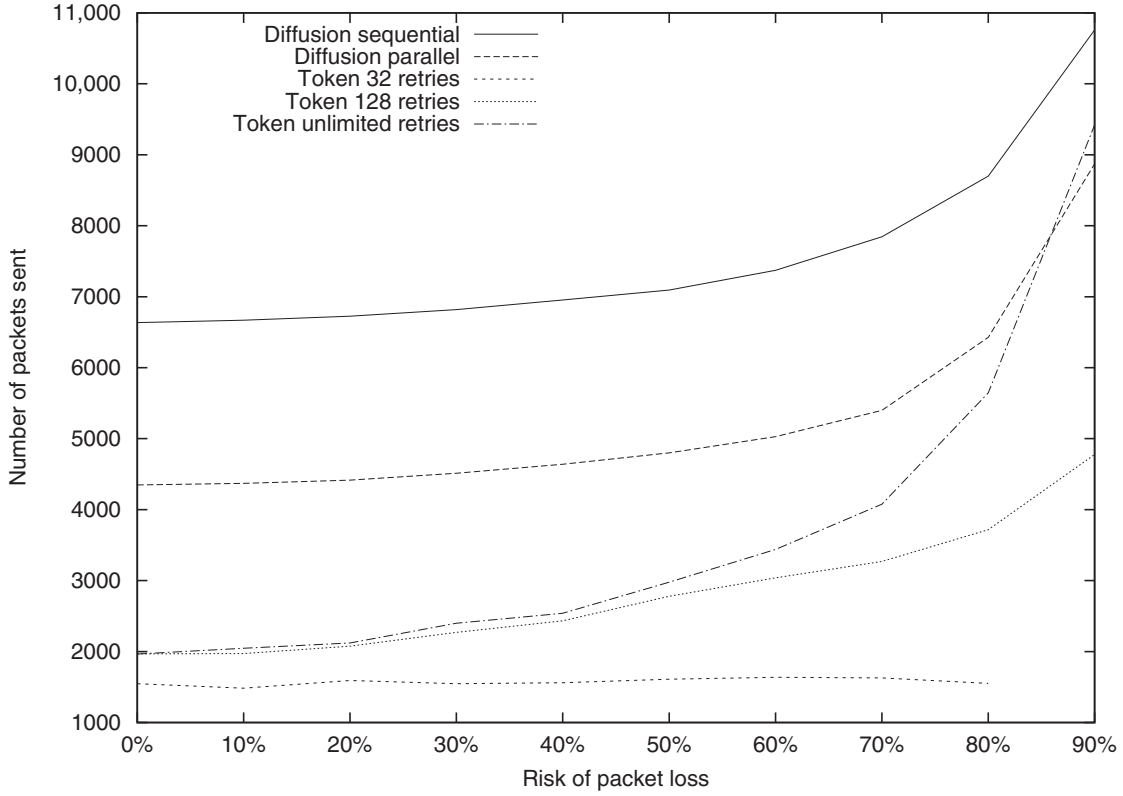


Fig. 10. Packet counts during self-reconfiguration at different rates of packet loss, corresponding to Fig. 8.

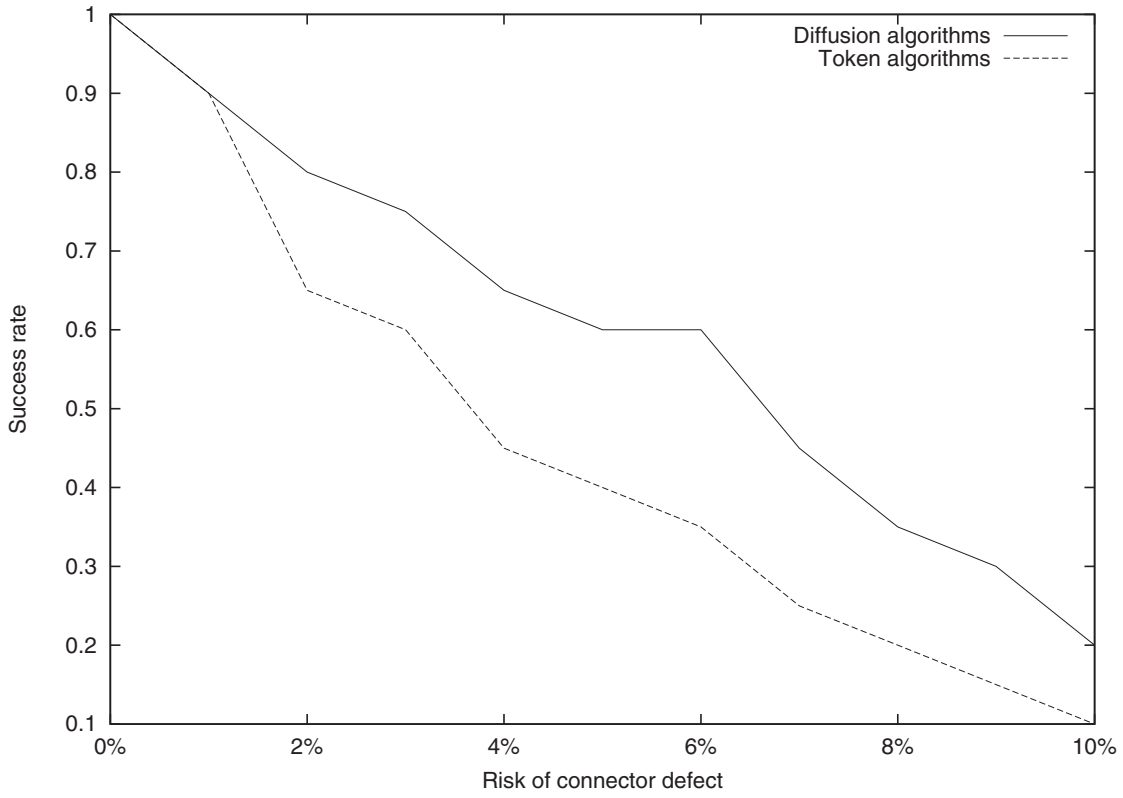


Fig. 11. Success rates at varying risks of defect communication connectors. Both diffusion-based and token-based algorithms have respective identical success rates.

that are highly unreliable, meaning that they exhibit very high degrees of packet loss. Similar to the previous experiment, we vary the risk that a given connector is unreliable, but since packets will eventually be transmitted even over unreliable connectors, we can measure the running time of the algorithm to give a more complete understanding of the performance of the algorithms on a system such as ATRON. Based on our experience with the physical ATRON modules, we set

https://doi.org/10.1017/S0263574710000664 Published online by Cambridge University Press

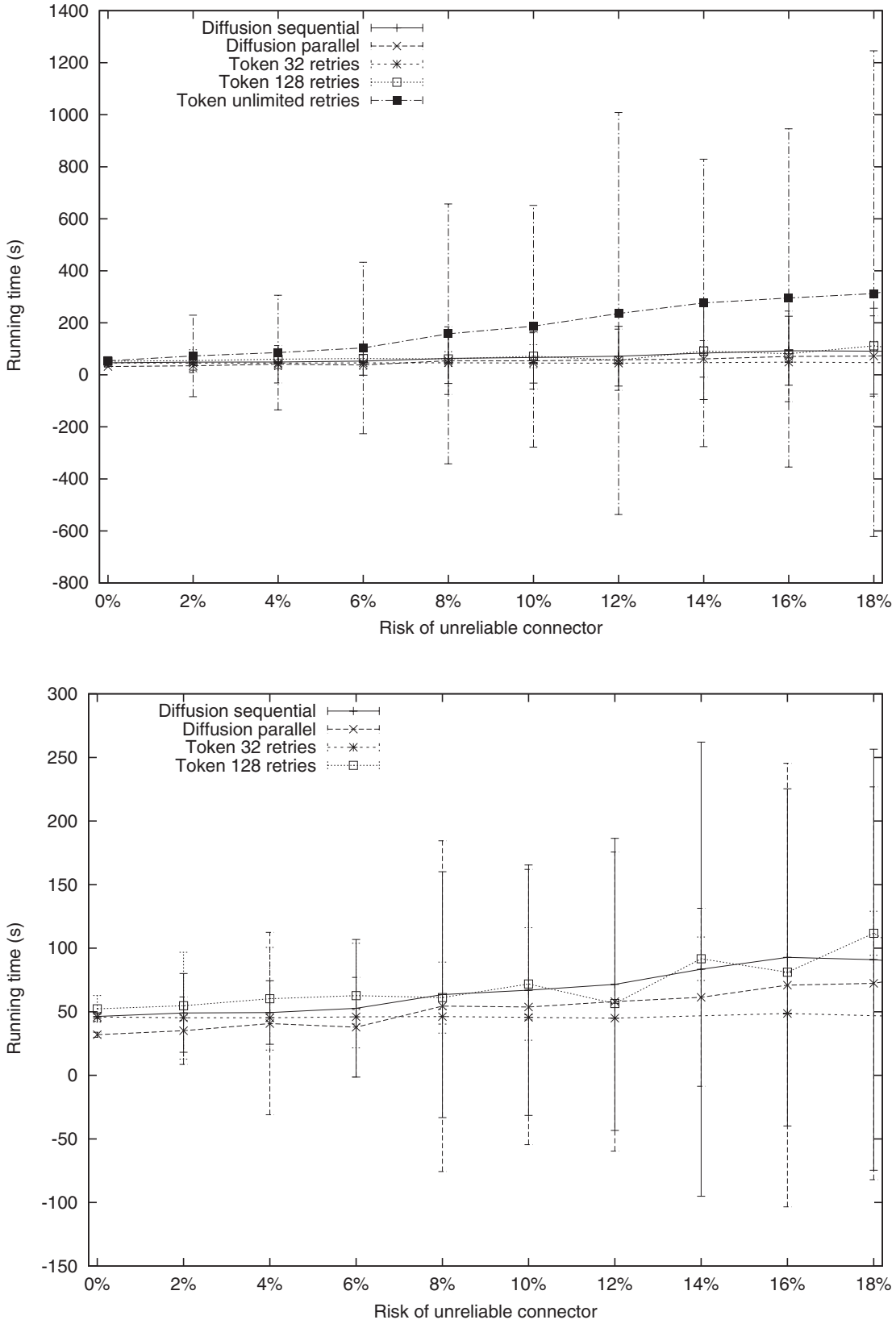


Fig. 12. Time for self-reconfiguration with 25% packet loss and varying risks of each connector being unreliable (bottom graph shows details of upper graph).

the general risk of packet loss at 25% and the risk of packet loss at unreliable connectors to 98% (keeping in mind that we sometimes observe a 20-second delay when transmitting packets between physical modules).

The running times of different algorithms at different rates of the risk of each connector being reliable are shown in Fig. 12, the upper graph shows an overall view, whereas the lower graph provides details not discernible in the overall

https://doi.org/10.1017/S0263574710000664 Published online by Cambridge University Press

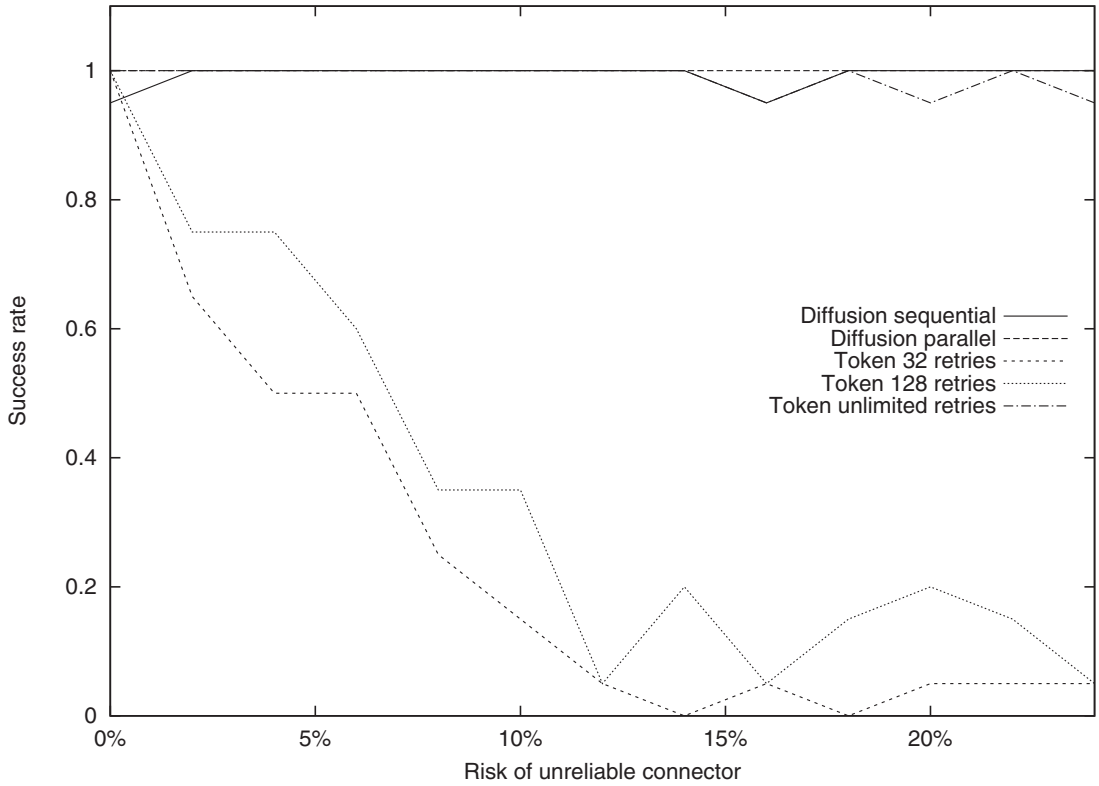


Fig. 13. Robustness of self-reconfiguration with unreliable connectors documents experiments of Fig. 12.

view. The success rates are shown in Fig. 13. As can be seen, the running time of the token-based unlimited retries increases massively, whereas the running times of the other token-based algorithms increase slowly, as is the case for the diffusion-based algorithms. Nevertheless, as can be seen from Fig. 13, which shows the success rates for the corresponding experiments, the running times for the token-based 32-retries and 128-retries algorithms are floor effects in the sense that they are much more failure-prone.

6.5. Reset tolerance

The reset tolerance on the physical ATRONs was demonstrated for our diffusion-based algorithm using a few simple experiments. These experiments demonstrate that the algorithm can correctly handle temporary unresponsiveness of a module, loss of state in an inactive module, and loss of state in an active module. Using the simulator we can perform a more thorough analysis of the reset tolerance of the diffusion-based algorithm, compared to the different variants of the token-based algorithm. We use the "8-to-car" self-reconfiguration sequence as before, but randomly reset module controllers to their initial state. A 2-second timeout is used in the first module of the sequence to detect if a new sequence is starting or a reset occurred in the middle of an existing one. Note that unlike the physical modules, reset does not influence any actuation currently taking place, we only investigate the tolerance toward loss of state. (Simulated interruption of rotation operations could be supported in the simulator by explicitly stopping the central actuator upon reset, whereas the simulator does not support stopping halfway through a connect or disconnect operation.)

Concretely, we measure the success rate of the "8-to-car" self-reconfiguration as a function of the probability of each module spuriously being reset at fixed-size intervals of roughly 1 s. Concretely, resets are desynchronized between modules by slightly offsetting the time interval by a 100-millisecond interval dependent on the module ID.† The results are shown in Fig. 14 with separate graphs for the token-based and diffusion-based algorithms. The three token-based algorithms exhibit similar performance, as is also the case for the two diffusion-based algorithms. The token-based algorithms quickly become ineffective as the risk of resets increases. On the contrary, the diffusion-based algorithms are highly robust toward resets, only starting to become ineffective at 94% risk of reset. We note that this degree of robustness could not be achieved with synchronized resets, as all state would be lost when all modules were reset at the same time (e.g., at 90% risk of reset per module every second, there would be a 48% risk of resetting all modules at the same time). However, given that the common causes of module resets are usually specific to the individual module and thus unrelated between different modules composing a structure, we argue that the probability of simultaneous resets rapidly decreases as a function of the number of modules.

† Precisely synchronized resets would represent the worst-case scenario, as multiple modules being reset at exactly the same time precludes restoring state from the corresponding modules. Desynchronized resets more closely model what one could expect from the physical modules.

https://doi.org/10.1017/S0263574710000664 Published online by Cambridge University Press

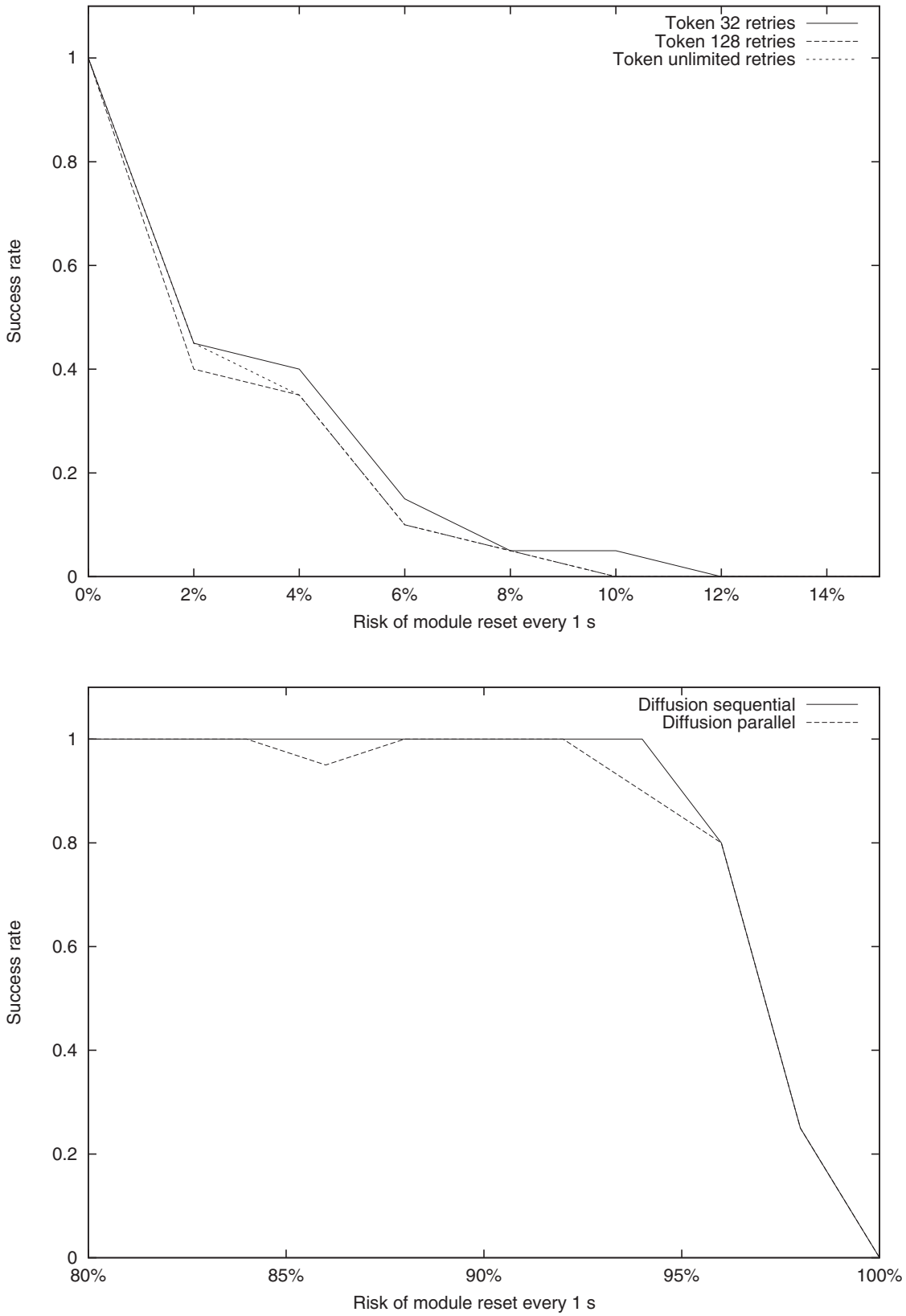


Fig. 14. Tolerance to spurious module reset: success rate as a function of individual module reset probability.

6.6. Morphology impact

To investigate the morphology impact of the robot, we have repeated the overall performance and reset tolerance experiments for a self-reconfiguration sequence that uses a snake morphology as target. Specifically, we start in the same configuration as before (the "eight" configuration) and self-reconfigure to a snake (as was shown in Fig. 4), then back again to the initial configuration; this sequence is repeated thrice to get an overall running time roughly equivalent to the "8-to-car" sequence.

https://doi.org/10.1017/S0263574710000664 Published online by Cambridge University Press

```
sequence rotate2independent {
  M__0.rotateFromToBy(0,216,true,150) &
  M__1.rotateFromToBy(0,216,true,150) &
  M__0.rotateFromToBy(216,0,true,150) &
  M__1.rotateFromToBy(216,0,true,150) ;
  (... repeated 2 more times)
}
```

```
sequence rotate2synchronized {
  M__0.rotateFromToBy(0,216,true,150) &
  M__1.rotateFromToBy(0,216,true,150) ;
  M__0.rotateFromToBy(216,0,true,150) &
  M__1.rotateFromToBy(216,0,true,150) ;
  (... repeated 2 more times)
}
```

Listing 3. Synchronized rotation experiment, parallel and sequential versions of two-module program.

The overall performance of the repeated "8-to-snake" self-reconfiguration sequence with 25% general packet loss and 98% packet loss on unreliable connectors is shown in Fig. 15 in terms of running time and success rate. In general, the performance is similar to what we observed for the "8-to-car" sequence. The 32-retries token-based algorithm is the fastest, but this is primarily due to a floor effect because very few sequences complete for high risks of unreliable connectors. The token-based unlimited retries, however, has a lower success rate than expected when compared to the "8-to-car" sequence due to a ceiling effect derived from the simulation time limit of 200 s. This ceiling effect is more acute for the repeated "8-to-snake" than for the "8-to-car" because a smaller set of connectors are being used for communication, which increases the risk that a large amount of communication is done through an unreliable connector.

The reset tolerance of the repeated "8-to-snake" self-reconfiguration sequence is shown in Fig. 16 for the token-based and the diffusion-based algorithms. The token-based algorithms behave roughly as for the "8-to-car" sequence, which is to be expected, as the sequence stops if the module holding the token is reset. The diffusion-based algorithms are effective up to 55% risk of reset (for the sequential sequence) and 80% risk of reset (for the parallel sequence); we expect that the difference is due to the shorter running time of the parallel sequence. Compared to the "8-to-car" sequence the reset tolerance is lower, probably due to the state spreading slower in the snake morphology after resets of multiple modules.

6.7. Scalability

The diffusion-based algorithm uses pending states to implement parallel operations. This approach has the disadvantage that synchronization of parallel operations can only be done globally by waiting for all parallel operations to complete. For larger ensembles of modules this requirement can become an overhead. The overhead is however largely dependent on the application, and moreover we expect that larger ensembles would be composed of multiple groups of modules, each executing their own scripts with their own shared state. Nevertheless, we can measure the overhead of global synchronization in a specific scenario, namely a long string of modules (in principle, a snake configuration) that rotate independently. For each experiment, we use two versions of the same program, exemplified for two modules in Listing 3. The program `rotate2independent` rotates the two modules first in one direction, then back again, all in parallel without any intermediate synchronization. The program `rotate2synchronized` performs the same operations but synchronizing all modules after the first rotation. Both programs repeat these steps thrice with an intermediate synchronization. The difference in running time between these two programs is thus the overhead of performing twice as many global synchronizations.

Table IV. Comparison of independent versus synchronized rotation of a string of modules.

$\#M$	Sequence	T	$\frac{T^s}{T^i}$	P	$\frac{P^s}{P^i}$	A	$C_i = \frac{P_i}{A_i T_i}$
1		5.9		0			
5	Independent	8.3		641		8	9.7
	Synchronized	9.7	1.17	749	1.17	8	9.7
10	Independent	10.6		1866		18	9.8
	Synchronized	12.7	1.20	2245	1.20	18	9.8
15	Independent	14.2		3947		28	9.9
	Synchronized	16.6	1.17	4600	1.17	28	9.9
20	Independent	17.5		6578		38	9.9
	Synchronized	20.6	1.18	7750	1.18	38	9.9

$\#M$: number of modules.
T: time in seconds to complete sequence.
P: number of packets.
A: number of active connectors

The running times for 5, 10, 15, and 20-module experiments with "rotating string of modules" are shown in Table IV, along with the base case of one module showing the time needed to complete the rotation operation. Only average running times over 20 experiments are shown; there is no packet loss or other noisy factors, and the standard deviation is negligible. As can be seen, the running time increases roughly linearly with the number of modules (more experiments would be needed to confidently determine if the increase is linear); this is however to be expected as communication time increases linearly with the number of modules due to the topology of the structure. We observe that the synchronization overhead remains roughly constant at a factor 1.2 over the independent version both in terms of running time (T) and packets sent (P).

The uniform topology used in the rotating string of module experiment allows us to more precisely analyze the sent number of packets. Modules always communicate their state to all their neighbors at a fixed rate, so the number of packets communicated per connector should not increase. Indeed, we can determine the number of packets sent per second per active connector, shown in the rightmost column of Table IV. This value, denoted as $C_{M\#}$ for a given number of modules i, is roughly the same across the number of modules and choice of sequence (more experiments would be needed to confidently determine if the value is constant). Packets are transmitted every 100 ms, or 10 times per second, which

https://doi.org/10.1017/S0263574710000664 Published online by Cambridge University Press

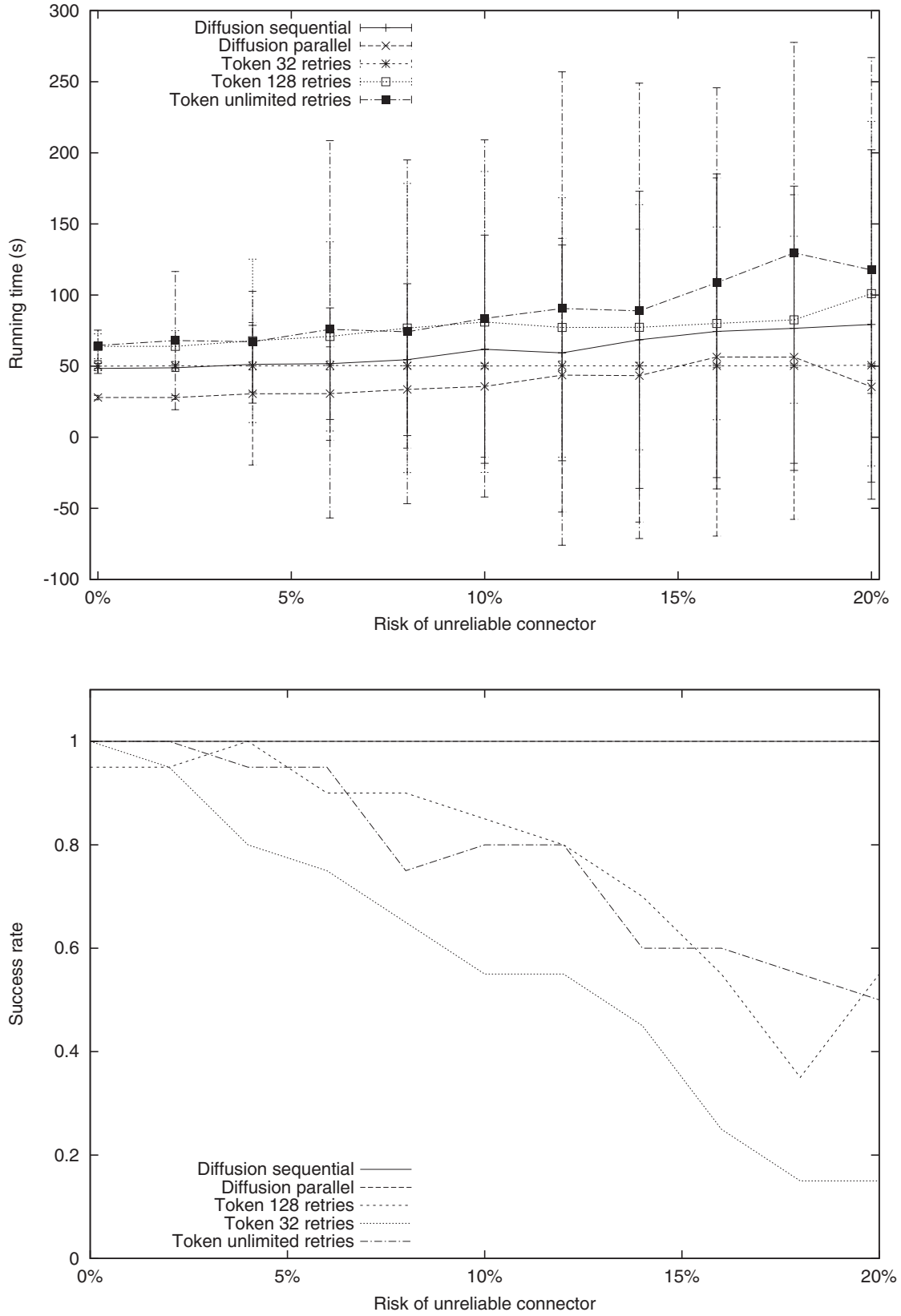


Fig. 15. Overall performance for snake morphology: 25% packet loss and varying risks of each connector being unreliable. Upper graph shows running times, lower graph shows success rates.

corresponds to the actual C value observed in simulation. Thus, adding more modules does not increase the amount of communication in the system, when measured per time-step per connector.

6.8. Assessment

The packet loss experiments demonstrate that the diffusion-based algorithm only has a significant advantage at very high rates of packet loss (Section 6.2) and

https://doi.org/10.1017/S0263574710000664 Published online by Cambridge University Press

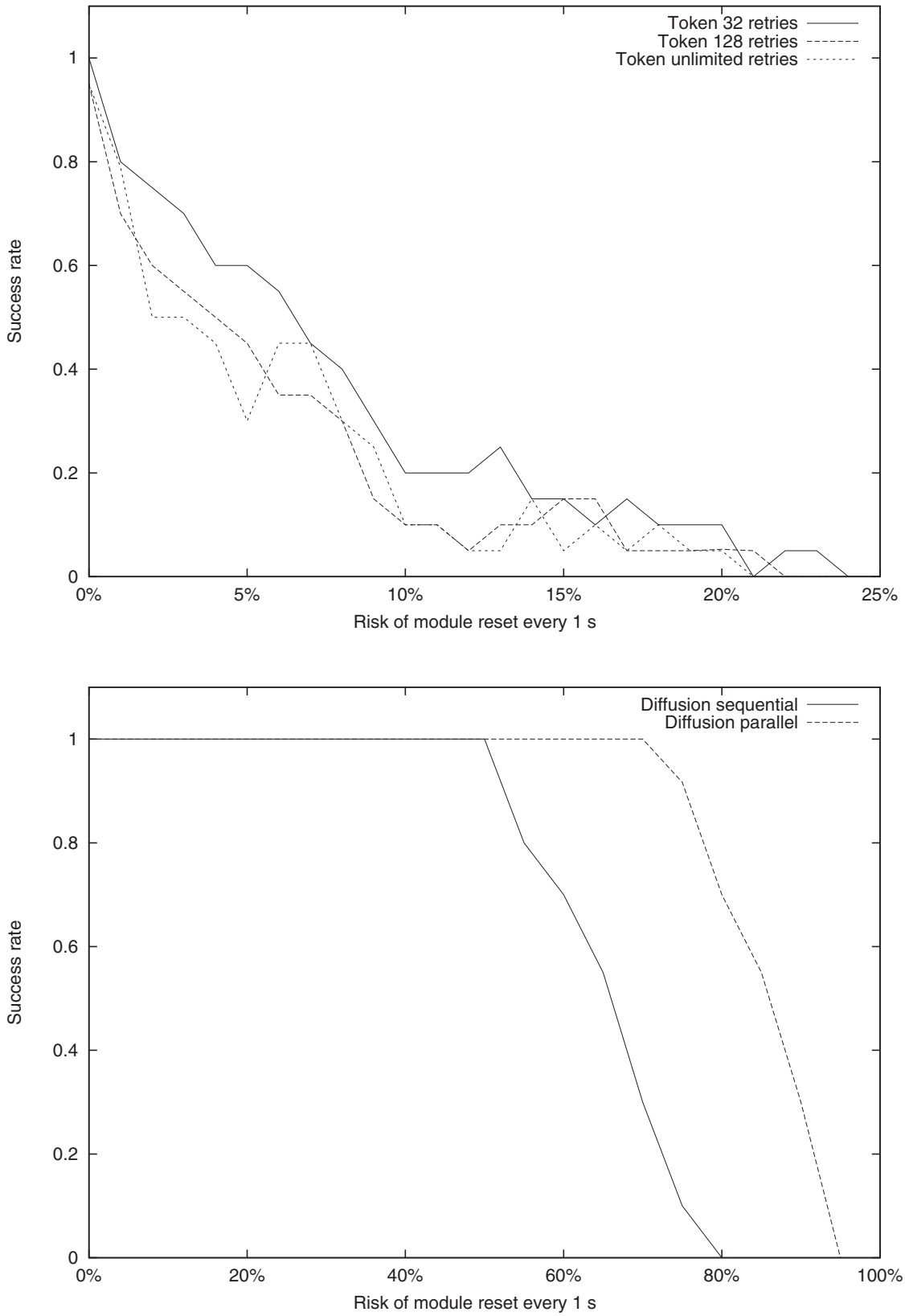


Fig. 16. Tolerance to spurious module reset when using the snake morphology: success rate as a function of individual module reset probability.

offers a significant albeit limited tolerance toward defect communication connectors (Section 6.3). This tolerance toward connections of variable communication quality is however a significant advantage in terms of overall performance in the case of heterogeneous connector sets combining fairly reliable and highly unreliable communication (Sections 6.4 and 6.6). We believe the latter case most closely models the actual physical ATRON hardware, and thus serves to support the observation that the diffusion-based algorithm provided a significant advantage

https://doi.org/10.1017/S0263574710000664 Published online by Cambridge University Press

on the physical modules during the entire "8-to-snake" sequence.

The spurious module reset experiments demonstrate that the diffusion-based algorithms offer a massive robustness advantage compared to the token-based algorithms (Sections 6.5 and 6.6), although the robustness is somewhat dependent on the morphology of the robot under highly unreliable conditions (Section 6.6). These results serve to demonstrate that the approach proposed in this paper provides a high degree of robustness toward loss of state, enabling, for example, aggressive use of watchdog timers to automatically restart unresponsive modules.

The scalability experiments demonstrate that global synchronization using pending states can have a limited, although significant, overhead on the running time and that the amount of communication remains constant per connector. We believe our state sharing and communication approach would scale to large number of modules, a claim that is supported, although by no means proven by these experiments. Programming large number of modules would, however, probably need a different high-level approach such as metamodules,[15] since scripting the individual movements of each module would not be feasible for large ensembles of unreliable modules. The scripting language proposed in this paper would nevertheless still be useful for programming individual steps, such as metamodule actions.

7. Conclusion

In this paper, we have shown how a distributed scripting language for self-reconfiguration can be compiled to run on the physical ATRON modules while providing both robustness toward partial failures and reversibility. Our system provides a significant improvement in terms of robustness compared to earlier efforts in our group, an improvement which we attribute to the use of a set of diversified tools to better address each aspect of the system. The use of specialized programming techniques for resource-constrained devices allows us to separate the concern of writing resource-efficient onboard control software and providing users with an expressive and easy to use high-level language.[5]

The immediate future work includes overcoming the limitations listed elsewhere in this paper, in particular the limitation on independent synchronization of parallel operations. On a longer term, we are interested in scaling to larger scenarios through the use of metamodules and dynamic reprogramming of modules. To program metamodules using the DynaRole, we would need to extend our approach to include a form of conditional selection of self-reconfiguration sequences so that metamodules can adapt their actions to their environment. Moreover, the communication protocol would need to be extended to restrict state-sharing to occur only between the modules that constitute the metamodule. Dynamic reprogramming of modules can be achieved by an integration with the DCD-VM,[7] which we believe is central to enabling a more agile and hence productive approach to working with self-reconfiguration.[5] The code distribution mechanism of the DCD-VM would, however, need to be extended in order to provide the same kind of robustness guarantees, as we present in this paper.

A key insight of our work is the explicit assumption that modules will exhibit partial failures and that our algorithms therefore must be designed to deal with this situation. As the mantra of modular robotics is to exploit physical redundancy to allow for simple, cheap, and possibly failing units, the underlying software should follow this approach as well. We expect that the use of these principles will enable even more complex experiments with modular robots in the future.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments, which helped to significantly improve the quality of our paper. We would also like to thank David Brandt for providing the original source code for the "8 to car" self-reconfiguration algorithm, which was the starting point for our implementation and moreover for providing useful background information on self-reconfiguration planning.

References

1. M. Asadpour, M. H. Z. Ashtiani, A. Sproewitz and A. Ijspeert, "Graph Signature for Self-Reconfiguration Planning of Modules with Symmetry," *Proceedings of the 2009 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, St. Louis, USA (2009).
2. M. Asadpour, A. Sproewitz, A. Billard, P. Dillenbourg and A. Ijspeert, "Graph Signature for Self-Reconfiguration Planning," *Proceedings of the 2008 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'08)*, Nice, France (2008) pp. 863–869.
3. M. P. Ashley-Rollman, S. C. Goldstein, P. Lee, T. C. Mowry and P. Pillai, " Meld: A Declarative Approach to Programming Ensembles," *Proceedings of the 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'07)*, San Diego, CA, USA (2007) pp. 2794–2800.
4. J.-C. Baillie, A. Demaille, Q. Hocquet, Nottale, M. and S. Tardieu, "The Urbi Universal Platform for Robotics," *Proceedings of the SIMPAR'08 Workshop on Standards and Common Platform for Robotics*, Venice, Italy (2008).
5. M. Bordignon, D. J. Christensen, K. Stoy and U. P. Schultz, "Elements of a Development Ecosystem for Modular Robot Applications," *Proceedings of the Fourth International Workshop on Software Development and Integration in Robotics (SDIR'09)*, Kobe, Japan (2009a).
6. M. Bordignon, L. Lindegaard Mikkelsen and U. P. Schultz, "Implementing Flexible Parallelism for Modular Self-Reconfigurable Robots," *Proceedings of the International Conference on Simulation, Modeling and Programming for Autonomous Robots (SIMPAR'08)*, Venice, Italy (2008) pp. 123–134.
7. M. Bordignon, K. Stoy and U. P. Schultz, "A Virtual Machine-Based Approach for Fast and Flexible Reprogramming of Modular Robots," *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'09)*, Kobe, Japan (2009b) pp. 4273–4280.
8. D. Brandt, "Comparison of A∗ and RRT-connect Motion Planning Techniques for Self-Reconfiguration Planning," *Proceedings of the 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'06)*, Beijing, China (2006) pp. 892–897.
9. D. Brandt, "Scalability and Complexity of Self-Reconfigurable Robot Control *PhD Thesis* (University of Southern Denmark: The Maersk Institute, 2007).
10. D. Brandt and D. J. Christensen, "A New Meta-Module for Controlling Large Sheets of ATRON Modules," *Proceedings*

https://doi.org/10.1017/S0263574710000664 Published online by Cambridge University Press

of IEEE/RSJ International Conference on Intelligent Robots and Systems, San Diego, CA (2007) pp. 2375–2380.
11. D. Brandt, J. C. Larsen, D. J. Christensen, R. F. Mendoza Garcia, D. Shaikh, U. P. Schultz and K. Stoy, "Flexible, FPGA-Based Electronics for Modular Robots," *Proceedings of the IROS'08 Workshop on Self-Reconfigurable Robots & Systems and Applications*, Nice, France (2008).
12. Z. Butler and D. Rus, "Distributed planning and control for modular robots with unit-compressible modules," *Int. J. Robot. Res.* **22**(9), 699–715 (2003).
13. D. J. Christensen, M. Bordignon, U. P. Schultz, D. Shaikh and K. Stoy, "Morphology Independent Learning in Modular Robots," *Proceedings of the International Symposium on Distributed Autonomous Robotic Systems (DARS'08)*, Tsukuba, Japan (2008a) pp. 379–391.
14. D. J. Christensen, D. Brandt, K. Stoy and U. P. Schultz, "A Unified Simulator for Self-Reconfigurable Robots," *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'08)*, France (2008b) pp. 870–876.
15. D. J. Christensen and K. Stoy, "Selecting a Meta-Module to Shape-Change the ATRON Self-Reconfigurable Robot," *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, Orlando, FL (2006) pp. 2532–2538.
16. P. Cohen, *Empirical Methods for Artificial Intelligence* (MIT Press, Cambridge, MA, 1995).
17. M. De Rosa, S. C. Goldstein, P. Lee, J. D. Campbell and P. Pillai, "Programming Modular Robots with Locally Distributed Predicates," *Proceedings of the 2008 IEEE International Conference on Robotics and Automation (ICRA'08)*, Pasadena, CA, USA (2008) pp. 3156–3162.
18. S. C. Goldstein, J. D. Campbell and T. C. Mowry, "Programmable matter," *IEEE Comput.* **38**(6), 99–101 (2005).
19. C. Intanagonwiwat, R. Govindan and D. Estrin, "Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks," *MobiCom '00: Proceedings of the 6th Annual International Conference on Mobile Computing and Networking*, New York, NY, USA, ACM Press (2000) pp. 56–67.
20. K. Klues, V. Handziski, D. Culler, D. Gay, P. Levis, C. Lu and Wolisz, A., "Dynamic Resource Management in a Static Network Operating System," *Tech. Rep.* WUCSE-2006-56 (Washington University in St. Louis, 2006).
21. S. Kokaji, S. Murata, H. Kurokawa and K. Tomita, "Clock synchronization algorithm for a distributed autonomous system," *J. Robot. Mechatronics* **8**(5), 317–338 (1996).
22. K. Kotay and D. Rus, "Algorithms for Self-Reconfiguring Molecule Motion Planning," *Proceedings of the International Conference on Intelligent Robots and Systems (IROS'00)*, Takamatsu, Japan (2000) pp. 2184–2193.
23. S. Murata, H. Kurokawa and S. Kokaji, "Self-assembling machine," *Proceedings of IEEE International Conference on Robotics and Automation (ICRA'94)* (1994) pp. 441–448.
24. E. Østergaard, K. Kassow, R. Beck and H. Lund, "Design of the ATRON lattice-based self-reconfigurable robot," *Auton. Robots* **21**(2), 165–183 (2006).
25. A. Pamecha, I. Ebert-Uphoff and G. S. Chirikjian, "Useful metrics for modular robot motion planning," *IEEE Trans. Robot. Autom.* **13**,531–545 (1997).
26. K. Prevas, C. Unsal, M. Efe and P. Khosla, "A Hierarchical Motion Planning Strategy for a Uniform Self-Reconfigurable Modular Robotic System," *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'02)*, Vol. 1, Washington DC (2002) pp. 787–792.
27. M. D. Rosa, S. Goldstein, P. Lee, J. Campbell and P. Pillai, "Scalable Shape Sculpting via Hole Motion: Motion Planning in Lattice-Constrained Modular Robots," *Proceedings of the 2006 IEEE International Conference on Robotics and Automation (ICRA'06)* (2006).
28. W.-M. Shen, B. Salemi and P. Will, "Hormone-inspired adaptive communication and distributed control for CONRO self-reconfigurable robots," *IEEE Trans. Robot. Autom.* **18**, 700–712 (2002).
29. K. Stoy, D. J. Christensen, D. Brandt, M. Bordignon and U. P. Schultz, "Exploit Morphology to Simplify Docking of Self-Reconfigurable Robots," *Proceedings of the International Symposium on Distributed Autonomous Robotic Systems (DARS'08)*, Tsukuba, Japan (2008) pp. 441–452.
30. C. Ünsal and P. Khosla, "A Multi-Layered Planner for Self-Reconfiguration of a Uniform Group of I-cube Modules," *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'02)*, Vol. 1, Maoui, Hawaii (2002) pp. 598–605.
31. C. Ünsal, H. Kiliccöte and P. K. Khosla, "A modular self-reconfigurable bipartite robotic system: Implementation and motion planning," *Auton. Robots* **10**, 23–40 (2001).
32. M. Yim, D. Goldberg and A. Casal, "Connectivity Planning for Closed-Chain Reconfiguration," *Proceedings of Sensor Fusion and Decentralized Control in Robotics Systems III*, Vol. 4196, Bellingham, WA, SPIE (2000) pp. 402–412.
33. M. Yim, W.-M. Shen, B. Salemi, D. Rus, M. Moll, H. Lipson, E. Klavins, and G. S. Chirikjian, "Modular self-reconfigurable robot systems [Grand Challenges of Robotics]," *IEEE Robot. Autom. Mag.* **14**(1), 43–52 (2007).
34. M. Yim, Y. Zhang, J. Lamping and E. Mao, "Distributed control for 3D metamorphosis," *Auton. Robots* **10**(1), 41–56 (2001).
35. T. Yokoyama, H. B. Axelsen and R. Glück, "Principles of a reversible programming language," *CF '08: Proceedings of the 2008 Conference on Computing Frontiers* (ACM, New York, NY, 2008) pp. 43–54.
36. E. Yoshida, S. Murata, A. Kamimura, K. Tomita, H. Kurokawa and S. Kokaji, "Motion Planning of Self-Reconfigurable Modular Robot," *Proceedings of the International Symposium on Experimental Robotics*, Honolulu, Hawaii (2000) pp. 375–384.
37. E. Yoshida, S. Murata, H. Kurokawa, K. Tomita and S. Kokaji, "A distributed method for reconfiguration of a three-dimensional homogeneous structure," *Adv. Robot.* **13**, 363–379 (1999a).
38. E. Yoshida, S. Murata, K. Tomita, H. Kurokawa and S. Kokaji, "An experimental study on a self-repairing modular machine," *Robot. Auton. Syst.* **29**, 79–89 (1999b).
39. Y. Zhang, A. Golovinsky, M. Yim and C. Eldershaw, "An XML-based Scripting Language for Chain-type Modular Robotic Systems," *Proceedings of the Conference on Intelligent Automous Systems (IAS-8)*, Amsterdam, Netherlands (2004) pp. 729–738.

https://doi.org/10.1017/S0263574710000664 Published online by Cambridge University Press