The 2009 IEEE/RSJ International Conference on
Intelligent Robots and Systems
October 11-15, 2009 St. Louis, USA

Robust and Reversible Self-Reconfiguration

Ulrik P. Schultz, Mirko Bordignon, Kasper Stoy.

Abstract— **Modular, self-reconfigurable robots are robots that can change their own shape by physically rearranging the modules from which they are built. Self-reconfiguration can be controlled by e.g. an off-line planner, but numerous implementation issues hamper the actual self-reconfiguration process: the continuous evolution of the communication topology increases the risk of communications failure, generating code that correctly controls the self-reconfiguration process is non-trivial, and hand-tuning the self-reconfiguration process is tedious and error-prone.**

To address these issues, we have developed a distributed scripting language that controls self-reconfiguration of the ATRON robot using a robust communication scheme that relies on local broadcast of shared state. This language can be used as the target of a planner, offers direct support for parallelization of independent operations while maintaining correct sequentiality of dependent operations, and compiles to a robust and efficient implementation. Moreover, a novel feature of this language is its *reversibility:* once a self-reconfiguration sequence is described the reverse sequence is automatically available to the programmer, significantly reducing the amount of work needed to deploy self-reconfiguration in larger scenarios. We demonstrate our approach with long-running (reversible) self-reconfiguration experiments using the ATRON robot and a reversible self-reconfiguration experiment using simulated MTRAN modules.

I. Introduction

Modular robotics is an approach to design, construction and operation of robotic devices aiming to achieve flexibility and reliability by reconfigurable assembly of simple subsystems [1]. Robots built from modular components can potentially overcome the limitations of traditional monolithic systems by rearranging their physical configuration on a need basis, a process known as self-reconfiguration, and by replacing unserviceable parts without disrupting the system's operations. Another potential advantage is the cost-effectiveness over more conventional robots, achieved through large-scale production of simpler, identical modules. This dictates economic and space constraints on both the mechanical and electronic design of the module units, favoring control hardware centered around simple microcontrollers and, in general, resource constrained embedded devices and dedicated hardware.

Self-reconfiguration is however difficult to implement in practice: the operations required to rearrange the modules from one physical shape to another must be determined, and the physical modules must correctly execute this sequence of operations. The former problem is key to self-reconfiguration and has been dealt with in the context of numerous different robotics systems [2], [3], [4], [5], [6], [7], [8]. The latter problem is the subject of this paper; it is in principle easy to solve but in practice involves many difficulties: real-world modules may have partial failures in their neighbor-to-neighbor communication abilities and may spuriously fail during the self-reconfiguration sequence. Moreover, the actual software implementation of the self-reconfiguration sequence is non-trivial. Even if the implementation is automatically generated by a planner, subsequent manual optimizations such as parallelization of operations [9] are non-trivial and significantly complicate the implementation since a partial ordering of self-reconfiguration operations typically must be maintained. As modular, self-reconfigurable robotic systems mature and gradually move into real-world applications [1], we believe the issue of robustness of self-reconfiguration will become more critical.

We are interested in enabling the programmer to quickly and reliably describe precise self-reconfiguration sequences that execute robustly on unreliable hardware. To this end, we have implemented a distributed scripting language that can execute self-reconfiguration sequences on the ATRON modular robot [10]. This scripting language is an extension to the DynaRole language [11] but provides a number of significant improvements. First, self-reconfiguration sequences are concisely described and compile to a robust and efficient implementation based on a distributed state machine. Second, dependencies between operations are explicitly stated allowing independent operations to be performed in parallel while maintaining an ordering between dependencies that are dependent on each other. Third, the language is *reversible* meaning that for any self-reconfiguration sequence the reverse sequence is automatically generated, which dependent on physical constraints makes any self-reconfiguration process described in the language reversible. We demonstrate the effectiveness of our approach with long-running, reversible self-reconfiguration experiments using physical ATRON modules and a reversible self-reconfiguration experiment using simulated MTRAN modules [2]. Reversible self-reconfiguration can be used in many cases since 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. We note that previous work with the DynaRole language used

This work was supported by the Danish Council for Technology and Innovation.

The authors are with the Modular Robotics Lab, Maersk Mc-Kinney Moller Institute, Faculty of Engineering, University of Southern Denmark, Denmark. {mirko,kaspers,ups}@mmmi.sdu.dk

[1] A trend witnessed and further stimulated by events such as the ICRA 2008 Planetary Contingency Challenge.

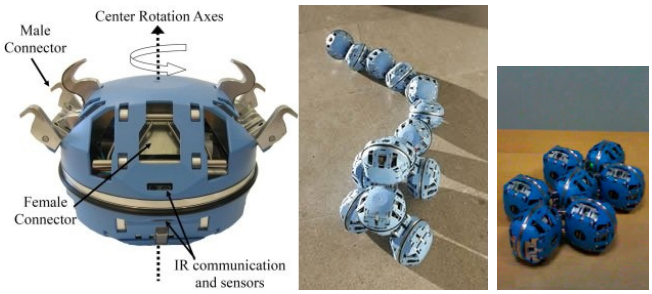

Fig. 1. An ATRON module (left) and various assembled structures (right)

the DCD virtual machine running on the physical modules as target [11]; in this paper we use a dedicated back-end to the compiler that generates either a TinyOS component or an ANSI C implementation. Robust distribution of bytecode and explicit support for blocking on split-phase operations would be required to use the DCD-VM but are not yet supported (see future work in Sect. VII for details).

The rest of this paper is organized as follows. First, Sec. II provides background information on the ATRON self-reconfigurable robot. Then, the main contribution is presented two parts: first Sec. III presents our embedded software platform for robust self-reconfiguration followed by Sec. IV 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 Sec. V. Last, we compare our contribution to related work in robotics and reversible computing (Sec. VI), and conclude with an overview of intended future work (Sec. VII). Our running example throughout the paper is self-reconfiguration of the ATRON robot from a flat "8-like" shape to a car shape, both shown in Fig. 1 along with a snake shape.

II. Background: The ATRON Robot

The ATRON self-reconfigurable modular robot (Fig. 1) is a 3D lattice-type system [10]. Each unit is composed of two hemispheres which rotate relatively to each other, giving the module one degree of freedom. Connection to neighboring 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 $\sim$20 MHz range and further 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 (e.g. sensor networks). The resulting low-level software implementation for the ATRON modules, described in more detail in [9], [11], [12], provides the foundations over which we developed the robust approach to self-reconfiguration, described in the next section.

III. 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. Moreover, the approach should allow parallel operations without violating sequential constraints (this is non-trivial 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 e.g. a probabilistic approach that provides robustness through redundancy. 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.

A. 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 IV). 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 using a robust communication protocol described later. A fragment of the source code of the state machine for the "8 to car" self-reconfiguration is shown in Fig. 2. 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 operations are at this level non-blocking and hence issue a `break` after starting a long-running operation, to ensure responsiveness of other tasks such as sharing the global state. The state machine could be written manually using the techniques described in this section, but we prefer to generate the implementation automatically, as described in the next section.

Given a protocol for globally sharing the state, a distributed state machine that uses a single 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

```
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;
```

Fig. 2. nesC fragment of the distributed state machine for "8 to car" (automatically generated)

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 completed, a given state can wait for the set of pending operations to be empty. This 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; supporting unrestricted dependencies is future work.

The handling of pending states can be seen in the source code fragment of the distributed virtual machine shown in Fig. 2. Here, state 1 on module `M0` starts a pending operation, adds 1 to the set of pending states, and transitions to state 2 which transfers control to state 4 on module `M3`. 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 state on a 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.

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

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

Fig. 3. 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 local state which will be propagated to all neighbors of the module.

		⟨state,address,pending⟩	
		`M0`	`M3`
	Initial state	$\langle 1, 0, \{\}\rangle$	$\langle 1, 0, \{\}\rangle$
1:	`M0` retract ⇒	$\langle 2, 0, \{1\}\rangle$	$\langle 1, 0, \{\}\rangle$
2:	`M0` transition ⇒	$\langle 4, 3, \{1\}\rangle$	$\langle 1, 0, \{\}\rangle$
	State propagate ⇒	$\langle 4, 3, \{1\}\rangle$	$\langle 4, 3, \{1\}\rangle$
4:	`M3` retract ⇒	$\langle 4, 3, \{1\}\rangle$	$\langle 5, 3, \{1\}\rangle$
	State propagate ⇒	$\langle 5, 3, \{1\}\rangle$	$\langle 5, 3, \{1\}\rangle$
3:	`M0` complete ⇒	$\langle 5, 3, \{\}\rangle$	$\langle 5, 3, \{1\}\rangle$
	Pending propagate ⇒	$\langle 5, 3, \{\}\rangle$	$\langle 5, 3, \{\}\rangle$
5:	`M3` continue ⇒	$\langle 5, 3, \{\}\rangle$	$\langle 6, 3, \{\}\rangle$

Fig. 4. Global state sharing for the "8 to car" example

B. Communication protocol

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, since 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 towards unstable communication where only a small percentage of messages get through.

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 complete. Each module continuously 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. 3. The two key properties that we exploit are (1) that a pending state p_0 added at 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 towards 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. This communication scheme is illustrated in Fig. 4 for the first few steps of the "8 to car" self-reconfiguration of Fig. 2. Module `M0` starts a pending operation which module `M3` waits for after having performed an operation. The global state sharing could be done in numerous other ways; we consider the evaluation of alternative approaches to be future work.

C. 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 or software errors. 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. Due to time constraints we have not yet implemented restart of pending operations in our system, but the following approach shows that it is feasible and likely to require just a small incremental improvement over our existing implementation. 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 reenter 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 a given module is 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 reentering the pending state will in many cases cause the pending operation to complete and hence provide the desired robustness.

D. 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 exploring other solutions to improve over this situation [13], at the moment we rely on compile-time virtualization of the hardware [14] and address the low communication performance using a 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 embodying 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 a 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 of 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 [9], and we are working on the integration of similar features in our virtual machine-based runtime implementation [11].

IV. Reversible Self-Reconfiguration

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

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

Fig. 5. DynaRole sequence describing the "8 to car" self-reconfiguration sequence

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 used directly by the programmer to quickly and concisely specify self-reconfiguration sequences or alternatively can 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.

A. Syntax and semantics

We have designed and implemented our distributed scripting language as an extension to the DynaRole language [11]. 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

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

TABLE I

THE DYNAROLE ATRON API, SELECTED OPERATIONS

in Fig. 5 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 I 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 [15]. A sequence of parallel statements are considered independent, that is, physically unconstrained, and may be executed in any order but must all 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 described earlier this feature is not supported by the current state machine design).

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 [11].

B. 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 Sec. II or alternatively an ANSI C implementation that can execute in the USSR simulation environment [16]. Concretely, the compiler generates an implementation of the distributed state machine described in Sec. III 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 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 and 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.

C. 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 currently relies on 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 for the API operations currently 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 Sec. VI for a discussion of reversal of non-API operations.

The programmer explicitly defines and invokes reversed sequences, for example the sequence shown in Fig. 5 is reversed using the following declaration:

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

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

D. 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, then 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 requires the programmer to only use 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.

V. Experiments

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

A. Physical ATRON modules

The first three experiments are performed using physical ATRON modules running code generated by our compiler from scripts written in DynaRole, and are illustrated in the video accompanying the submission. The first experiment serves as an overall documentation that our compiler works, can generate reverse sequences, and that it 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 is running the "8 to car" self-reconfiguration sequence implemented by the program of Fig. 5. 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; the last sequence is documented by the accompanying video. 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 completes 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. 7) and power off

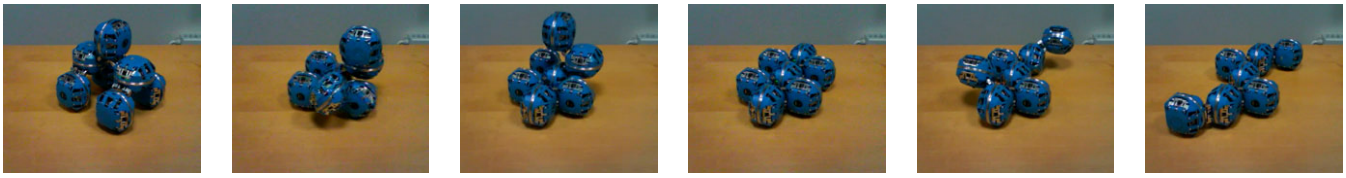

Fig. 6. Self-reconfiguration from car to snake; the self-reconfiguration process subsequently reverses and returns the robot to the car shape (not shown)

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

Fig. 7. Robustness towards 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

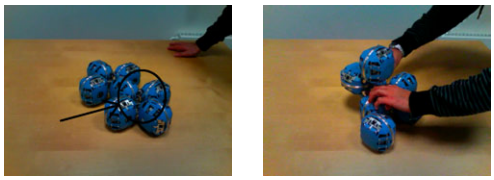

Fig. 8. Left: robustness towards an active module being reset, the highlighted module is reset while opening a connector. Right: robustness towards failing communication demonstrated by manually blocking communication paths in turn.

a module while it is performing a connection operation (Fig. 8, left). In both cases the self-reconfiguration process continues 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 does the self-reconfiguration continue (Fig. 8, right). These experiments are also documented by the accompanying video. Thus, we have demonstrated a tolerance towards 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. 6, we refer to the accompanying video for the reverse sequence. 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.

B. 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 [16]. 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. 9; as was the case for the ATRON robot the reverse sequence is automatically derived.

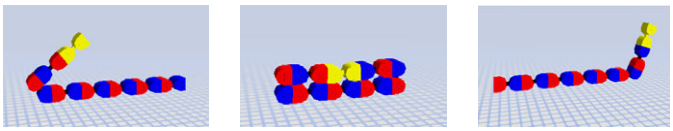

Fig. 9. Reversible reconfiguration with simulated MTRAN modules: displacing a module from the left to the right is manually programmed whereas the reverse self-reconfiguration is automatically derived.

VI. Related Work

Off-line planning of self-reconfigurable robots has been studied for a large number of different robotic systems [3], [7], [8], [17], [18]. These approaches are largely complementary to our work: any off-line planner (for the ATRON robot) could use our distributed scripting language as output and would thus benefit from the robustness and reversibility. The runtime execution of the self-reconfiguration process on physical robots is treated in some cases; Brandt for example uses a token-based approach where a single token that is explicitly routed through the module structure is used to represent the active module [19]. This approach is efficient but less robust since it does not tolerate partial failures.[2]

Dynamic planning approaches typically address the runtime execution of the self-reconfiguration process [2], [4], [5], [6]. These approaches however typically assume working two-way communication; this assumption is essential for on-line algorithms that self-reconfigure the robot depending on the current shape but is not required in our case since self-reconfiguration is carried out according to an off-line plan. Robustness is explicitly investigated by Yoshida et al

[2]Modifying the implementation generated by Brandt by replacing the token-based approach with the global state sharing approach was experimentally determined to provide a massive advantage in terms of robustness, which was the initial inspiration for our work.

using the Fracta system, 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 [20], [21].

General-purpose programs can be made reversible, as demonstrated by Tetsuo et al for the general-purpose high-level language Janus [22]. Janus allows general algorithms such a Fast-Fourier Transform to be implemented and automatically reversed. This approach is much more general than our highly restricted, distributed scripting language which for example does not have a representation of state and does not have control structures. Nevertheless, as demonstrated by Tetsuo 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 Tetsuo et al has been highly inspirational for making a reversible language for self-reconfiguration.

VII. Conclusion and Future Work

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 towards 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, like specialized programming techniques for resource-constrained devices to provide a reliable runtime and high-level languages to provide ease of use and expressiveness [12]. Moreover, the explicit assumption that modules will exhibit partial failures and our algorithms therefore must be designed to deal with this situation was a key factor in approaching the problem from what we believe is the right perspective: 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.

The immediate future work includes overcoming the limitations listed elsewhere in this paper. A central interest is in a more complete integration with the DynaRole language and DCD-VM, which we believe is central to enabling a more agile and hence productive approach to working with self-reconfiguration [12].

Acknowledgements: We would 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 insights on self-reconfiguration planning.

References

[1] 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. Automat. Mag.*, March 2007.

[2] E. Yoshida, S. Murata, H. Kurokawa, K. Tomita, and S. Kokaji, "A distributed method for reconfiguration of a three-dimensional homogeneous structure," *Advanced Robotics*, no. 13, pp. 363–379, 1999.

[3] A. Pamecha, I. Ebert-Uphoff, and G. S. Chirikjian, "Useful metrics for modular robot motion planning," *IEEE Transactions on Robotics and Automation*, no. 13, pp. 531–545, 1997.

[4] C. Ünsal, H. Kiliccöte, and P. K. Khosla, "A modular self-reconfigurable bipartite robotic system: Implementation and motion planning," *Autonomous Robots*, no. 10, pp. 23–40, 2001.

[5] Z. Butler and D. Rus, "Distributed planning and control for modular robots with unit-compressible modules," *The International Journal of Robotics Research*, no. 22, pp. 699–715, 2003.

[6] 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," in *Proc. of the 2006 IEEE Int. Conf. on Robotics and Automation (ICRA'06)*, 2006.

[7] K. Kotay and D. Rus, "Algorithms for self-reconfiguring molecule motion planning," in *Proc. of the Int. Confe. on Intelligent Robots and Systems (IROS'00)*, 2000.

[8] E. Yoshida, S. Murata, A. Kamimura, K. Tomita, H. Kurokawa, and S. Kokaji, "Motion planning of self-reconfigurable modular robot," in *Proc. of the Int. Symp. on Experimental Robotics*, 2000.

[9] M. Bordignon, L. Lindegaard Mikkelsen, and U. P. Schultz, "Implementing Flexible Parallelism for Modular Self-Reconfigurable Robots," in *Proc. Int. Conf. on Simulation, Modeling and Programming for Autonomous Robots (SIMPAR'08)*, Venice, Italy, 2008.

[10] M. W. Jørgensen, E. H. Østergaard, and H. H. Lund, "Modular ATRON: Modules for a self-reconfigurable robot," in *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS'04)*, Sendai, Japan, 2004.

[11] M. Bordignon, K. Støy, and U. P. Schultz, "A Virtual Machine-based Approach for Fast and Flexible Reprogramming of Modular Robots," in *Proc. IEEE Int. Conf. on Robotics and Automation (ICRA'09)*, Kobe, Japan, May 12-17 2009.

[12] M. Bordignon, D. J. Christensen, K. Støy, and U. P. Schultz, "Elements of a Development Ecosystem for Modular Robot Applications," in *Proc. of the Fourth Int. Workshop on Software Development and Integration in Robotics (SDIR'09)*, Kobe, Japan, May 12 2009.

[13] D. Brandt, J. C. Larsen, D. J. Christensen, R. F. Mendoza Garcia, D. Shaikh, U. P. Schultz, and K. Støy, "Flexible, FPGA-Based Electronics for Modular Robots," in *Proc. of the IROS'08 Workshop on Self-Reconfigurable Robots & Systems and Applications*, Nice, France, September 22 2008.

[14] K. Klues, V. Handziski, D. Culler, D. Gay, P. Levis, C. Lu, and A. Wolisz, "Dynamic Resource Management in a Static Network Operating System," Washington University in St. Louis, Tech. Rep. WUCSE-2006-56, 2006.

[15] J.-C. Baillie, A. Demaille, Q. Hocquet, M. Nottale, and S. Tardieu, "The Urbi Universal Platform for Robotics," in *Proc. SIMPAR'08 Wksh. on Standards and Common Platform for Robotics*, Venice, Italy, Nov. 3 2008.

[16] D. J. Christensen, D. Brandt, K. Støy, and U. P. Schultz, "A Unified Simulator for Self-Reconfigurable Robots," in *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS'08)*, France, 2008.

[17] D. Brandt, "Comparison of A$*$ and RRT-connect motion planning techniques for self-reconfiguration planning," in *Proc. of the 2006 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS'06)*, Beijing, China, Oct. 2006, pp. 892–897.

[18] M. Asadpour, A. Sproewitz, A. Billard, P. Dillenbourg, and A. Ijspeert, "Graph signature for self-reconfiguration planning," in *2008 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS'08)*, September 2008, conference, pp. 863–869.

[19] D. Brandt, "Scalability and complexity of self-reconfigurable robot control," Ph.D. dissertation, The Maersk Institute, University of Southern Denmark, Sept. 2007.

[20] E. Yoshida, S. Murata, K. Tomita, H. Kurokawa, and S. Kokaji, "An experimental study on a self-repairing modular machine," *Robotics and Autonomous Systems*, vol. 29, pp. 79–89, 1999.

[21] S. Kokaji, S. Murata, H. Kurokawa, and K. Tomita, "Clock synchronization algorithm for a distributed autonomous system," *Journal of Robotics and Mechatronics*, no. 8, pp. 317–338, 1996.

[22] T. Yokoyama, H. B. Axelsen, and R. Glück, "Principles of a reversible programming language," in *CF '08: Proc. of the 2008 conference on Computing frontiers*. New York, NY, USA: ACM, 2008, pp. 43–54.