Robotica (2018) volume 36, pp. 625–654. © Cambridge University Press 2018
doi:10.1017/S0263574717000613

Modelling reversible execution of robotic assembly

Johan Sund Laursen*, Lars-Peter Ellekilde and Ulrik Pagh Schultz

The Maersk Mc-Kinney Moller Institute, University of Southern Denmark, Denmark.
E-mails: lpe@mmmi.sdu.dk, ups@mmmi.sdu.dk

(Accepted December 13, 2017. First published online: January 11, 2018)

SUMMARY
Programming robotic assembly for industrial small-batch production is challenging; hence, it is vital to increase robustness and reduce development effort in order to achieve flexible robotic automation. A human who has made an assembly error will often simply undo the process until the error is undone and then restart the assembly. Conceptually, robots could do the same. This paper introduces a programming model that enables robot assembly programs to be executed in reverse. We investigate the challenges in running robot programs backwards and present a classification of reversibility characteristics. We demonstrate how temporarily switching the direction of program execution can be an efficient error recovery mechanism. Moreover, we demonstrate additional benefits arising from supporting reversibility in an assembly language, such as increased code reuse and automatically derived disassembly sequences. As a default approach to reversibility, we use program inversion and statement-level inversion of commands, but with a novel override option providing alternative sequences for asymmetric reverse actions. To efficiently program for this model, this paper introduces a new domain-specific language, SCP-RASQ (Simple C++ Reversible Assembly SeQuences). In initial experiments, where 200 consecutive assemblies of two industrial cases were performed, 18 of 22 errors were corrected automatically using only the trial-and-error capabilities that come from reverse execution.

KEYWORDS: Robot software architectures; Domain-specific programming language; Reversible computing; Error recovery; Assembly and disassembly.

1. Introduction

Increasing the flexibility of robot setups allows small and medium sized enterprises (SMEs) to install and use robots for more varied tasks and allow greater penetration of robots into SME-like manufacturing with small-batch and low-volume productions. This is considered an important task for future industrial development.[1,2] Robot assembly and manufacturing is, however, challenged by the uncertainties associated with the parts, the surroundings, and the system itself. These uncertainties may generate errors and the implementation of error avoidance measures can become both complex and costly. We believe that many such errors can be solved automatically using a trial-and-error approach, and that focusing on error correction and recovery reduces the amount of custom-made error avoidance code required.

To enable efficient use of our error correction approach, we create a reversible programming language for robot assembly. In this language, a program can be executed forward to perform assembly and backwards to perform disassembly. Reversibility naturally facilitates a backtracking and retrying approach to handle errors.

By integrating reverse execution into the robot programming language, we explore a novel synergy between reversible computing and robotics. Concretely, two hypotheses are investigated:

Regarding robustness, we hypothesise that errors related to stochastic effects during assembly operations can be efficiently addressed using reverse execution of the program, allowing the robot to

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

https://doi.org/10.1017/S0263574717000613 Published online by Cambridge University Press

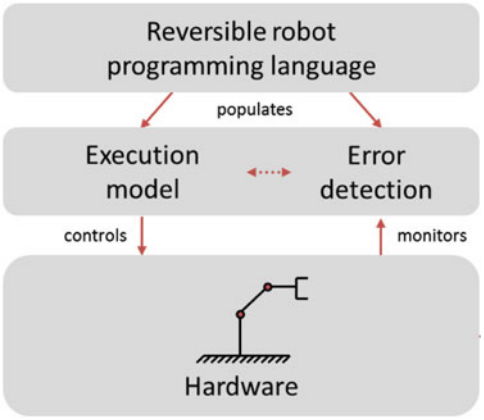


Fig. 1. Reversible-language models the reversible executions overall structure and specifies error detection.

temporarily back out of an erroneous situation, after which the assembly operation can be automatically retried. This allows the system to automatically resolve many critical issues without explicitly written error handlers.

Regarding disassembly, we hypothesise that reverse execution can facilitate programming of assembly operations that can be used for both assembling and dissembling an object. Reverse execution could enable automatic derivation of certain required operations from their forwards counterparts, and more generally an entire disassembly sequence could be derived from an assembly sequence, or vice versa.

An appropriately defined concept of reverse execution could further provide benefits in the form of increased software reuse through invocation of the same sequence of operations both forwards and backwards and in general make programming easier, as less time has to be spent on calibrating and fine-tuning program parameters. Reverse execution could be used for solving errors in cases where there are uncertainties or other small variations associated with the process. For example, grasping an object repeatedly may result in slightly different positions for the object due to the stochastic nature of the operation. Errors occurring as a result of this nondeterminism cause probabilistic behaviour, in that the outcome of the operation may change each time. Such errors can frequently occur in assembly sequences. For instance, when attempting to screw in a bolt where the thread may not catch the counterpart correctly. By unscrewing and removing the bolt, and then repeating the original sequence of instructions, the orientation may have changed slightly, enabling the bolt to catch the threads correctly, allowing the assembly to proceed.

Traditional control-based systems actively address these uncertainties through the use of sensors, e.g., as described by Jörg *et al.*[3] Vision-based control systems for dealing with uncertainties in assembly was suggested by Zhang *et al.*[4] and Neto *et al.*,[5] whereas Chen *et al.*[6] applies a methodology where the system actively searches for solutions. Passive solutions, on the other hand, use simulation with added uncertainties to find robust and uncertainty-tolerant trajectories.[7] Koval *et al.* demonstrated a similar passive approach.[8] The approach presented in this paper is different in that it does not explicitly model the uncertainties of the assembly task, but relies on probabilistic effects to eventually handle errors rather than trying to avoid them.

An overall structure of the reversible framework and programming language developed can be seen in Fig. 1. This shows the underlying software and execution model for achieving reversible programs, the high-level SCP-RASQ language and the error detection module. The framework is tested with our robotic platform[7] shown in Fig. 2 together with the two industrial assembly cases used throughout this paper.

A key contribution of this paper is in terminology development for reversible programming of assembly tasks. The terminology is based on the analysis and classification of reversibility in real assembly tasks, robots, and reversibility in related subjects. The analysis and terminology are used to create a formal model of reversible robot assembly programming. To facilitate the programming of reversible assemblies, this paper presents the SCP-RASQ domain-specific language for writing reversible assembly programs. We use SCP-RASQ to demonstrate the applicability and error handling

https://doi.org/10.1017/S0263574717000613 Published online by Cambridge University Press

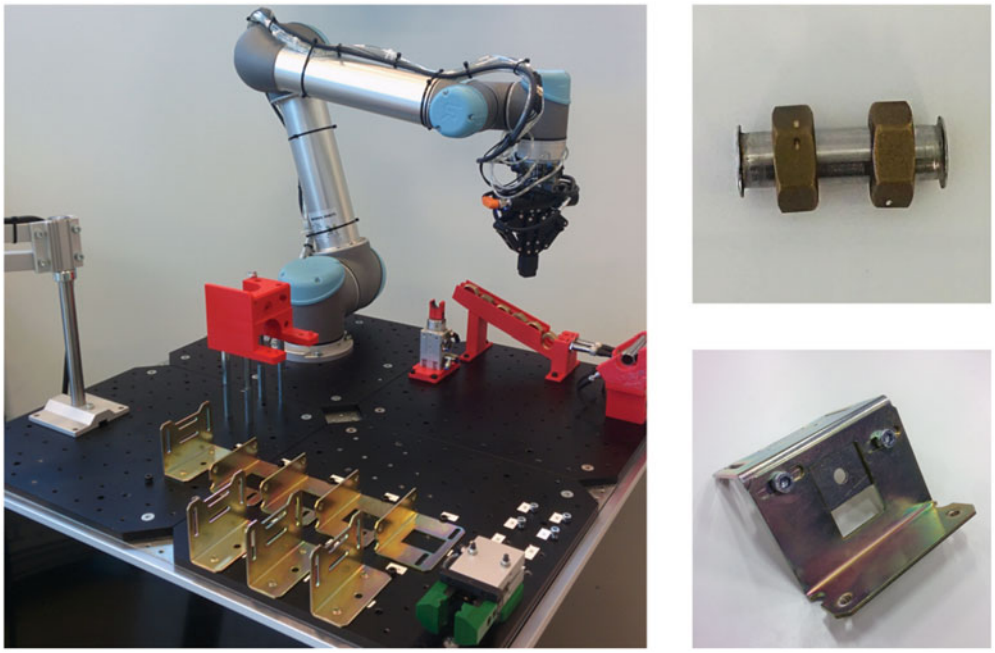


Fig. 2. The experimental platform and two assembly test cases.

mechanisms derived from reversible assembly programs. The underlying model also provides an XML-based representation, which allows interaction with other languages and high-level planners.

Our model employs program inversion[9] as the default mechanism for reversibility. Unlike existing approaches to program inversion, users are, however, allowed to override the default reverse implementations and provide explicit reverse behaviours. This allows for *indirectly reversible* operations, which are performed differently forward and backwards. As an example, pushing an object away is different from pulling the object, as the latter requires a grasp on the object: simply inverting instructions is not always sufficient for reversing operations. Our model of reversibility is created to fit domain-specific abstractions of assembly tasks. Assembly is assumed to have an overall sequential flow of operations, which may be a collection of complex reversible operations and instructions. The model lets users create reversible programs for assembly and provides a reversible interface to the physical hardware.

Connection to previous work: In a previous work, we describe a high-level reversible DSL syntax and language for robot assembly.[10] We also lay out the groundwork for a model of how to do reversible assembly.[11] This paper presents a structured analysis of reversible assembly. This is used to refine and simplify the model and expose additional benefits, such as the ability to both call functions forward and backwards.[9] Additional related work and experiments validating the use of reversible execution is also presented.

Paper organization: The rest of this paper is organised as follows: Section 2 discusses related work. Section 3 defines the concept of reversibility in assembly tasks. Section 4 discusses programming and software modelling of reversible assembly. Sections 5–7 describe various aspects of our implementation of the reversible robot framework, respectively the software components marked *execution model*, *reversible robot programming language,* and *error detection* in Fig. 1. Section 8 documents our experiments, and Section 9 presents our conclusions and outlines future work.

2. Related Work and Background

Our efforts in designing a reversible programming language for robotic assembly are rooted in problems associated with small-batch assembly and automation (Section 2.1). To achieve the desired robustness and reliability in the automation setups, we use a reversible model for efficient error handling (Section 2.2). Reversible assembly tasks are programmed through a domain-specific language related to those already commonly used for programming robots (Section 2.3). For the

https://doi.org/10.1017/S0263574717000613 Published online by Cambridge University Press

implementation and understanding of a model of reversibility, we draw on inspiration from reversible computing and general-purpose reversible programming languages (Section 2.4). Finally, we address how reversibility is handled in current robot controllers and setups (Section 2.5).

2.1. Precision and small batch assembly

Currently, small-batch assembly using robots is rarely feasible. Robotic assemblies are challenged by uncertainties from sensors, robot kinematics, and tolerances on materials. Several approaches are being researched that use different means of overcoming these uncertainties and increasing the performance of the systems. Thomas *et al.*[12] utilises active compliance. Buch *et al.*[13] relies on simulation and passive compliance and Neto *et al.*[5] uses vision-based sensing. However, regardless of the approach used operations may fail and give errors. Whether used as a stand-alone-tool or in combination reverse execution can provide a simple tool to further improve robustness and performance.

2.2. Error handling

According to the classifications by Loborg,[14] the process of error handling is commonly split into three different subtasks: error detection, error classification, and error recovery. Our proposed framework addresses the problem of error recovery. We discuss error detection and classification in a reversible context but our main focus is not in the area of error detection. Our system, currently, uses a manually specified error detection mechanism. The default backtracking error recovery strategy can, however, be executed at any given time and mixing this with a more generic error detection could therefore further increase the feasibility of the system. Generic solutions inspired by the immune system, such as those proposed by Christensen *et al.*,[15] Tarapore *et al.*,[16] or Canham *et al.*,[17] assume that faults change the flow of sensory data. The system can be trained to recognize abnormalities using learning and pattern recognition, meaning that this could eliminate the need for users to manually specify error conditions. Alternative systems based on vision and spatial reasoning about assembly cases could be applied, as described by Okumura *et al.*[18] or Wang *et al.*[19]

Error recovery in automation and robot systems tends to either be implemented through knowledge or graph-based systems. Knowledge-based approaches use pre-defined actions to recover from errors.[20] Graph-based system searches the graph for another path that sidesteps the error. Even though our approach to error-recovery is graph-based, it does not rely on graphs with the same complexity as those often found in state-tracking Markov chains or petri-net graphs.[21]

Definitions by Loborg[14] regarding error recovery in automation systems classifies our approach as applying backwards error recovery. The system attempts to reach a previously passed error-free state by undoing its latest operations. The backwards approach to error recovery should be seen in contrast to forward error recovery, where the system attempts to reach an error-free state through alternative forward actions and re-planning strategies. Our current strategy is naive—we backtrack a step and then attempt forward execution. If this does not work, we backtrack another step and try again. Suggestions for more sophisticated solutions are discussed and outlined in Section 7.3.3.

Donald addresses error detection and recovery strategies.[22] He tackles the consequences of dealing with errors for which there is no planthat guarantees that the task can be accomplished. He states that two choices exist: giving up, or making strategies that may work but fail reasonably when they cannot. This is also called failure dependency and the principle is discussed as graceful degradation.[1] Creating programs that fail gracefully and with reasonable errors is key to making the reversible system work with the backwards recovery strategy.

2.3. Robot programming

In contrast to general-purpose languages such as Java and C++, domain-specific languages (DSLs) are designed to facilitate programming in a specific domain and/or given a particular set of conditions.[23] Designing robotic programming languages is both an applied practice and an ongoing field of research. Several examples of proprietary robot languages exist. Kuka Robot Language (KRL)[24] and ABB's Rapid language[25] are typical examples of such proprietary languages. The languages are used for programming industrial manipulators, but in general do not assume anything about the applications. In the literature, several attempts and suggestions for introducing new programming techniques and existing general-purpose paradigms into robotic programming frameworks can be found.[26] The literature also contains several domain-specific languages targeting specific applications domains.[27] Examples include languages for pick and place,[28] assembly,[29] control,[12] and perception.[30]

https://doi.org/10.1017/S0263574717000613 Published online by Cambridge University Press

A commonly used design for current assembly languages is to rely on action and skill libraries, e.g., as suggested by Huckaby *et al.*[31] or by Wahrburg *et al.*[32] These provide a layered architecture and rely on domain abstractions for handling complex operations and control flow. The advantage of the design is that users can create programs using existing primitives and experts can add new complex primitives as long as they comply with the framework's conditions, as shown by both Thomas *et al.*[12] and Bøgh *et al.*[33] Our framework employs a similar structure.

This paper describes two interfaces for programming assembly tasks using our reversible framework: An internal C++ DSL and an XML-based syntax that allows other software components to interface the framework. This interface could allow integration of more advanced programming languages or allow the framework to serve as a target for high-level and symbolic planners. High-level intelligent planners such as those based on the approach of Fikes and Nilsson[34] or more recently that of Garrett *et al.*[35] could be used to create an overall strategy for the assembly. Our framework is, thus, not a competitor for these high-level approaches, but should rather be viewed as a potential target—which could increase robustness and promote synergetic effects.

This paper does not address the topic of object feeding. Improvements in generic, fast and easy to setup feeding technologies would greatly benefit the commercial feasibility of small-batch automation. As demonstrated by both Hoffmann *et al.*[36] and Edmonson *et al.*,[37] feeding is an active research topic, and solutions such as bin-picking are becoming commercially available.

2.4. Reversible computing

Our approach to reversible execution in assembly languages is inspired by reversible general-purpose programming languages. Reversible programming languages are typically based on the principle of *backtracking*[38] in a state space or *program inversion*.[39] Systems using backtracking record decisions, variables, and state during forward execution. Reverse execution is then achieved by backtracking from memory. Systems using program inversion can interpret programs both forward and backwards. Backtracking works without altering the syntax of the program, which makes it ideal for debugging tools, such as that suggested by Agrawal *et al.*[40] Program inversion provides more flexibility and allow programs to be executed backwards without an initial forward execution.

The principle of program inversion was presented by Tetsuo *et al.*,[9] where it was used in the design of the general-purpose programming language, Janus. They demonstrated how general-purpose programs could be made reversible, and implemented a reversible Fast-Fourier Transform algorithm to showcase the language. While we use the same core principle for achieving reversible assembly programs, our approach is fundamentally different. Janus is designed around the concept of an entirely reversible language. This means leaving out nonreversible instructions (e.g., simple boolean operations such as *and* and *or* operations) in order to avoid compromising reversibility. In contrast, our approach to reversibility is as a practical feature for error recovery to help ease the programming of robot assembly tasks. This means inclusion of both reversible and nonreversible concepts, and that reverse execution might not return to the initial state. It also means that while Yokoyama *et al.* demonstrate how general-purpose language structures, control flow, and calculation can be reversed, they do not address the issues of connecting the language to external I/O and other nonreversible outside influences. This remains an open issue, and we believe the work presented in this paper can be a useful contribution in this direction.

2.5. Reversibility in robots

Simple reversibility features in robot languages already exist and have been shown to have commercial value. Most mainstream controllers and embedded robot programming languages include simple and limited reverse execution. In these languages, the feature is used when programming and debugging the robots.[41] Manufacturers use various approaches and implementations: for example, KRL offers different options, one of which is backtracking. Execution is recorded during forward motion and can be backtracked afterwards.[24] Fanuc uses a simple implementation of program inversion based on backward interpretation.[42] This allows the user to step through programs in reverse order, but only operates on the robots' move commands and cannot reverse control-flow structures. RAPID adds to this approach, allowing users to specify alternative reverse instructions for backward execution of a subroutine.[43] Any of these implementations are useful for reversing the movements of the robot. They are, however, not suited for conceptually reversing execution of entire assembly tasks. We propose a setup where reversibility is deeply integrated into the programming process. Programs are

https://doi.org/10.1017/S0263574717000613 Published online by Cambridge University Press

created to be capable of automatically executing direction switches during run-time upon encountering errors. This is in contrast to the commercial systems, where users are involved in determining if backward steps are valid each time the feature is used. Note that while we use reversibility in robot controllers for inspiration, we are not trying to replace these languages. These target general-purpose robot programming, our framework is more specialized and only targets programming of reversible assembly tasks.

Schultz *et al.*[44] showed how reverse execution could be applied within modular self-reconfigurable robots. Here, reverse execution was applied to automatically derive a reverse self-reconfiguration sequence from a forwards one, enabling the robot modules to automatically reconfigure themselves back to an earlier configuration, thus reducing the work of the programmer since only one direction has to be programmed. This work was later extended to a more general reversible language for controlling self-reconfigurable robots performing locomotion tasks.[45]

2.6. Disassembly using robots

We apply reverse execution for the dual purpose of error recovery and automatic disassembly. The field of disassembly is an entire research field with a major focus on disassembly of partially unknown objects that relies on cognitive robotics.[46] Vongbunyoung *et al.* provide a comprehensive introduction to automated disassembly[47] and numerous cases dealing with disassembly can be found in the literature. Szalatkiewicz addresses the potential recycle value of electronic equipment by analysing the outcome of a disassembled hard-disk.[48] Scharke highlights many of the obstacles of implementing flexible and automated disassembly after products end-of-life.[49] Our approach to disassembly is, however, not rooted in recycling but based on executing a known assembly program in reverse and is thus fundamentally different to these existing approaches. We apply reversibility to achieve robustness in manufacturing by enabling backtracking. Potential benefits could also be as a tool to facilitate debugging of robot programs or increased software reuse through invocation of the same sequence of operations for both forwards and backwards execution.

3. Reversibility in Assembly Tasks

Executing advanced robotic assembly sequences involves operations such as precise placement of objects, insertions with tight fits, screwing operations, and so forth. All are challenged by uncertainties from sensors, robot kinematics, and part tolerances. This section outlines definitions and concepts regarding reversibility in assembly tasks in the real world. The results are based on the inspection of 13 real industrial small-batch assembly cases that are further described in Section 3.1. The following section (Section 4) addresses the concept of reversibility in software models, computation, and programming of assembly tasks.

3.1. Reversibility of real assembly tasks

A collection and description of small-batch industrial assembly cases can be found in our technical report.[50] The report describes 13 cases, which consist of a total of 71 suboperations. The cases come from various companies and represent workpieces that are currently assembled manually. We observe that all of the 13 cases consist of purely sequential flow. While none of the cases are distinctively nonreversible, eight cases contain nonreversible operations. However, in 5/8 of these cases, the only nonreversible operation is the very last and final operation that completes the assembly.

Table I lists and enumerates the assembly operations that can/cannot be reversed. We do not, however, have full knowledge of the cases, only the description in the report, so numbers should only be viewed as estimates. We observe that while several cases have operations that are nonreversible, this does not mean the concept of backtracking cannot be applied to the remaining operations.

3.2. Basic terms

Based on the cases described in Section 3.1, we define the following terms regarding subsets of assembly as follows. An *assembly task* is subdivided into operations. *Operations* can be further subdivided into instructions or nested operations. *Instructions* are further distinguished into primitives and actions. Operations describe the high-level logic of the case in a functional manner. Instructions, in turn, are closely connected to the hardware. *Primitives* are commonly used hardware-related commands such as moving to a position. *Actions* are complex and user-defined commands requiring

https://doi.org/10.1017/S0263574717000613 Published online by Cambridge University Press

Table I. Estimated reversibility of assembly operations from 13 industrial cases.

Reversible	54/71	76.1%
Place	23	32.4 %
Screw	18	25.4 %
Put	8	11.3 %
Other	5	7.0 %
Unknown	8/71	11.3%
Mount	5	7.0 %
Other	3	4.3 %
Nonreversible	9/71	12.7%
Press	6	8.5 %
Activate	3	4.2 %

sensor fusion and control loops. Reversible robotics needs to work on both an operation and instruction level.

3.3. Reversibility: doing and undoing

Many physical phenomena and actions are in principle reversible, although this reversibility may depend on the abstraction level at which they are observed. Robotics represents a real-world application area where computational reversibility has a physical counterpart. Conceptually, several kinds of actions performed by robots can be reversed. For example, driving a mobile robot from one room to another and back again or reversing an industrial assembly process to perform disassembly are both conceptually reverse tasks, but very different in complexity. Here, the presence of a physical environment adds a layer to what it means to be reversible. Reversing the semantic meaning of an operation may require a different set of instructions compared to its forwards counterpart. This means that even if each instruction is reversible in itself, the resulting operation is not necessarily practically reversible. For example, the IO-commands that activates an external screwdriver might be different from the commands that put it in reverse. Therefore, the principle of program inversion is not sufficient to achieve physical reversibility.

In order to deal with the different kinds of reversibility in robot automation, we need to distinguish conceptually reversible and nonreversible assembly operations.

We further divide the conceptually reversible operations into two categories: *directly reversible* and *indirectly reversible* operations. Operations that can be reversed through the default approach of program inversion are considered directly reversible. Indirectly reversible operations, on the other hand, can be reversed, but requires a different sequences of instructions. An example is the inverse of pushing an object, which is very different from pulling it, even though the two operations can be considered inverses. Regardless of the desire to support reversibility, an assembly program may include a mix of types, and must therefore have the capabilities to distinguish between them.

To summarize, we distinguish between three types:

Directly reversible: Reversible by running the inverse of each instruction in a sequence in reverse order.
Indirectly reversible: Reversed by another sequence of instructions.
Nonreversible: Cannot be reversed.

3.4. Repeatability: doing and redoing

A most straightforward form of repeatability is found in reversible computer programs, such as those written in Janus. These programs are said to be *time-invertible*. This means that the program is independent on how many times, and in which directions, it has been executed previously. Reversible computing programs are normally deterministic and will for a given input yield the same result each time. Programs interacting with an outside environment, such as robot programs, are however

https://doi.org/10.1017/S0263574717000613 Published online by Cambridge University Press

Table II. Classification of assembly tasks based on reversibility and repeatability.

Repeatability	Reversibility	Example
Full	Direct	Pick and place
	Indirect	Pushing and pulling
	None	Painting
Partial	Direct	Screwing
	Indirect	Inserting nails
	None	Drilling
None	None	Welding

different. Reversibility may depend upon and change based on external effects and previously executed operations.

Physical changes made to the environment from the execution influence the repeatability of operations. Moving a box from A to B and then back again can be done over and over without changing the environment and thereby the outcome of the operation. These operations are defined as *fully-repeatable*. Other actions such as inserting and removing a screw can only be done a limited number of times due to wear and tear. These are said to be *partially-repeatable*. Finally, *nonrepeatable* operations cannot be retried, e.g., inserting nails or doing click-fit assemblies.

Repeatability is not necessarily defined in terms of the number of times an operation can be repeated. While screwing operations can be repeated more times than the insertion of nails, gluing operations are more influenced by the hardening time. This work assumes that repeatable assembly tasks are either repeatable or nonrepeatable. A simple theoretical model of how to handle repeatability was proposed by Schultz *et al.*,[10] which allowed users to manually specify the number of repeats associated with various tasks. A more general and flexible model is, however, left as future work.

Combining the concept of reversibility and repeatability, we classify different assembly operations as shown in Table II. Note that the categorisation is dependent on interpretation and implementation, but nonetheless provides a general idea of how operations are different with regard to reversibility.

4. Programming Reversible Assembly

We now address the concept of reversible programming and how to create a software model of reversible assembly. This is complicated by the need to address reversibility both in the context of assembly tasks, the software model, and in the interplay between the two.

We are interested in reversible physical robotic assembly tasks rather than a theoretical model. In other words, we wish to have robots performing realistic and practically useful assembly sequences. While the proposed framework is for doing reversible assembly, we make no assumptions regarding the reversibility of the underlying hardware. Our approach allows integration of existing and external equipment (robots, feeders, presses, or other machines) into our framework.

4.1. Flow and subdivision of assembly tasks

Based on our observations regarding flow, subdivision and reversibility, a formal software model for the programming of reversible assembly tasks is defined. Figure 3 shows our model of the subdivision of assembly tasks. The model also shows the constructs used to model reversible flow. Regarding reversible flow, we make the following distinctions:

Assembly tasks: Comprised of a sequential flow of operations.

Operations: Requires reversible flow operators to map instructions and other operations into the high-level assembly case logic.

Instruction: Provides a two-way reversible forward/backward mapping of hardware instructions. These are programmed using traditional nonreversible programming. Alternatively, instructions provide a nonreversible single directional mapping.

https://doi.org/10.1017/S0263574717000613 Published online by Cambridge University Press

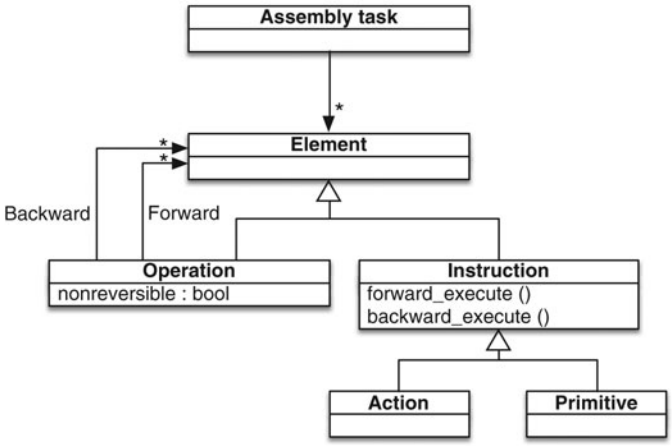


Fig. 3. UML model of conceptual reversible assembly tasks with reversible flow constructs.

The distinctions are based on the following observations: (1) Overall assembly cases tend to have sequential control flow. As they are created for repetition, they are void of runtime decisions since this makes systems significantly more complex. (2) Operations enable modelling of both reversible and nonreversible behaviour along with various other reversible properties using the principles of directly and indirectly reversible flow. (3) Instructions map the nonreversible interfaces of external hardware into reversible two-way primitives or nonreversible primitives.

While reversibility of operations can be described by reversible control flow constructs, the underlying tools, machines, and robots used to perform the assembly cannot be assumed to be reversible in either performance or interface. Primitives are used to define easily reversible mappings of hardware commands. Actions allow for complicated (and often very case-specific) nonreversible algorithms to be packaged in a reversible fashion, thereby maintaining the reversibility of the system as a whole. Examples are control loops, complex algorithms, and limitations due to sensors. Concepts such as visual-servoing or applying a force/torque sensor may have no obvious reverse. In many cases, the action may, however, be reversible, but requires additional parameters or a different syntax.

4.2. Reversible robot control capabilities

Our approach uses statement-level program inversion as the underlying principle for reverse execution. This gives a general and flexible approach for modelling assembly programs. The result is a framework model where programs can be interpreted both forward and backwards. Different capabilities are, however, used to provide this behaviour. This section outlines some of the relevant methods, some of which are however not used in our model. The two basic methods are:

Forward/backwards interpretation: If a program is executed backwards, the program pointer is decreased at each step. This means the original sequence of instructions is read in reverse when interpreting the code backwards.

Instruction inversion: Each primitive instruction has different semantics for forward and reverse. This means instructions behave differently when executed in forward or backwards direction. In reversible general-purpose languages, this corresponds to replacing plus with minus and multiplication by division.

Using either of these two methods can in simple cases be sufficient to achieve the desired degree of reversibility. Using the two methods in combination does, however, provide a good basis for achieving reversible programs. We further distinguish the following specialized reversible programming concepts:

Reversible control flow: For reversing conditionals such as `if`, `while`, and `for`. This requires a reversible notion of conditions in addition to that found in traditional programming.[9]

Reversible subroutine calls: For reversing unconditional jumps. This is often associated with procedural calls and functions.

Overridden reverse flow: Allows users to write different code for forward and backwards execution.

https://doi.org/10.1017/S0263574717000613 Published online by Cambridge University Press

While the two first principles are central to the field of reversible computing they are seldom implemented in robot programming languages. The principle of overridden reversible flow is, however, a concept not used in reversible computing, but which is already found in some robot controllers.[25]

Reversible programming languages do not address the issues of connecting the language with persistent data storage and other external I/O, but handle the subject of control flow and language structures. Reversibility in commercial robotics, on the other hand, uses a very pragmatic approach for interfacing to the real world but do not address the control flow and language structures required to achieve a fully reversible robot language. In this paper, we merge the two principles to achieve a reversible language for assembly, which integrates with the real world, real machines, and provides the reversible flow constructs needed. In this regard, this paper contributes a useful notation of external I/O to the field of reversible computing.

We do not currently apply the concept of *reversible conditional flow operators*. Although useful in general-purpose languages, they are not essential for modelling reversible assembly behaviour as long as individual instructions are written to conform to the principle of *instruction inversion*.

4.3. Reversible instructions

The principle of instruction inversion implies certain restrictions on which instructions can be executed and how. These restrictions are not always satisfied in the interface available to control the hardware. In our approach, each instruction has an inverse, e.g., an *open gripper* command executed backwards becomes a *close gripper* command and a *switch I/O on* becomes a *switch I/O off*.

Depending on the use-case and interface, we can classify normal instructions (generally found in robot programs) into the following different use-case patterns:

One-to-one: A command that changes the state from a known state to a new known state. Toggling I/O commands follow this structure. Toggling an I/O to off indicates the previous state was on.

Many-to-one: A command that changes the state of the workcell from an unknown to a known state. For example, a move command tends to follow this structure, and will move from any current position to the specified joint configuration.

One-to-many: Changes the state of the workcell from a known to an unknown state. Activating force compliance mode instructs the robot to apply a force in a given direction and as such does not have a clear specified end target. Often the end target is implicit and provided by physically limiting the movement of the robot.

Many-to-many: Moving from an unknown starting position to a visually-servoed end position. The actual location of the end position may move with time, which is irreversible.

The use of *many-to-one* instructions is common in normal interfaces, but pose problems for reverse execution. This problem is illustrated in Fig. 4, where the target of a move command is assumed to be the same in both directions. At the top is a typical sequence for placing an object. The bottom figure shows how the gripper is activated in an incorrect position if executed backward, which motivates the need for different semantics for asymmetric instructions.

In conventional robot language interfaces, most instructions work as *many-to-one* instructions, but for statement-level program inversion to work, instructions must follow the *one-to-one* pattern. In these instructions, the state of the platform can be deduced both before and after execution. Therefore, for our language to feature the full range of instructions normally found in simple robot languages, and for the program inversion principle to work, we modify the commands such that they follow a *one-to-one* pattern. For instance, a typical move command would normally be specified with only one argument in the form of *move to A*. In our model, the command is provided with two arguments in the form *move from B to A*. From this, a reversible relative motion *move* Δq can be deduced.

Rather than relying on the user specifying instructions according to the *one-to-one* pattern, we use categorization and automatic classification of instructions to determine how instructions are to be reversed. This approach is further described in Section 5.5.

4.4. Implicit and persistent instructions

It is common for robot programs to contain instructions that have no effect and do not change the overall state of the workcell. Conversely, some instructions continue to effect the state of the workcell after they have been executed. We distinguish the following two categories, as they influence how we address reversibility.

https://doi.org/10.1017/S0263574717000613 Published online by Cambridge University Press

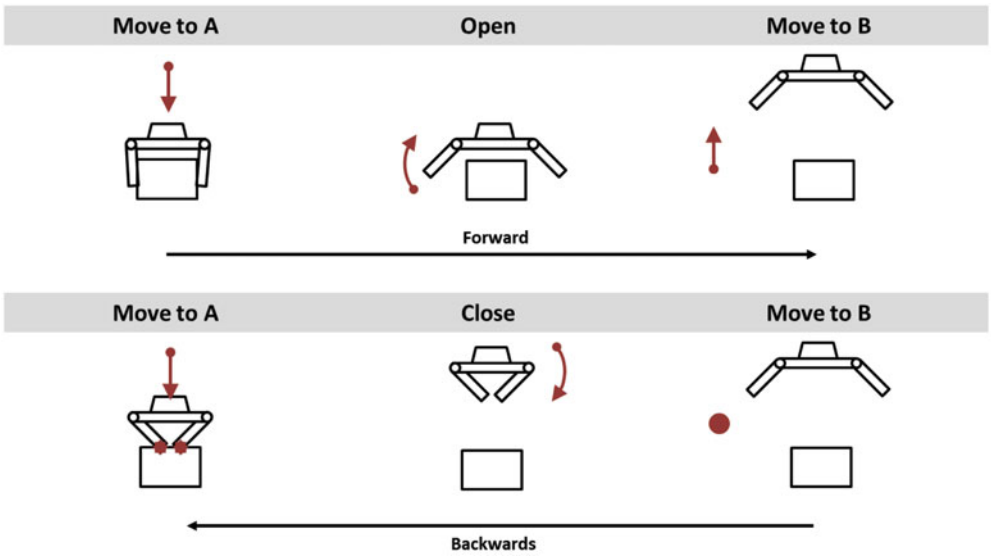


Fig. 4. Example of how the position where the gripper is activated may change depending on the direction of execution, unless the *one-to-one* semantics of the move instructions are taken into account.

Implicit condition: To ensure that a grasp is not attempted with a closed gripper, the first command in a sequence will often be to open the gripper. This assumes the gripper was closed when the function was called. This may, however, not be the case and the command ends up not changing the state of the workcell.

Persistent: Activates a continuously running effect while continuing execution of the program. The activated effects may impact the state of the system. This is often seen when external equipment is activated, such as turning on a screwdriver while continuing to move the robot or changing speed and acceleration of the robot.

Reversing a command that works as an implicit condition may break reversibility. Therefore, the same principles used to map instructions to follow the *one-to-one* pattern are also used to identify instructions that work as implicit conditions. These instructions can then be dealt with appropriately.

4.5. Summary

Figure 5 shows relations and connections between the discussed terms. The figure shows our subdivision of assembly task as consisting of operations, which again is made by instructions. It then relates the fundamental principles used to achieve reversibility for both individual instructions and multiple instructions concatenated into an operation before connecting concepts, which more specifically changes or influences the principles. Note that if pure forward/backward interpretation and instruction inversion are used directly reversible assembly is obtained, whereas the use of reversible flow modifiers can create indirectly reversible behaviour.

5. Execution Model

We now define a model for execution of assembly tasks. The execution model is our underlying implementation and representation of the reversible assembly program. Afterwards, we introduce our domain-specific language, which is used to mimic the constructs of reversible assembly. Model and language together share responsibility for implementing the reverse effects needed.

5.1. Model definition

We propose an execution model built on the following concepts:

1. Instructions follow the *one-to-one* pattern as they have different semantics for forward and reverse.
2. Indirect reversibility is achieved by modelling instruction sequences that are different for forwards and reverse execution using the principle of overridden reverse flow.
3. Instructions can be marked as nonreversible.

https://doi.org/10.1017/S0263574717000613 Published online by Cambridge University Press

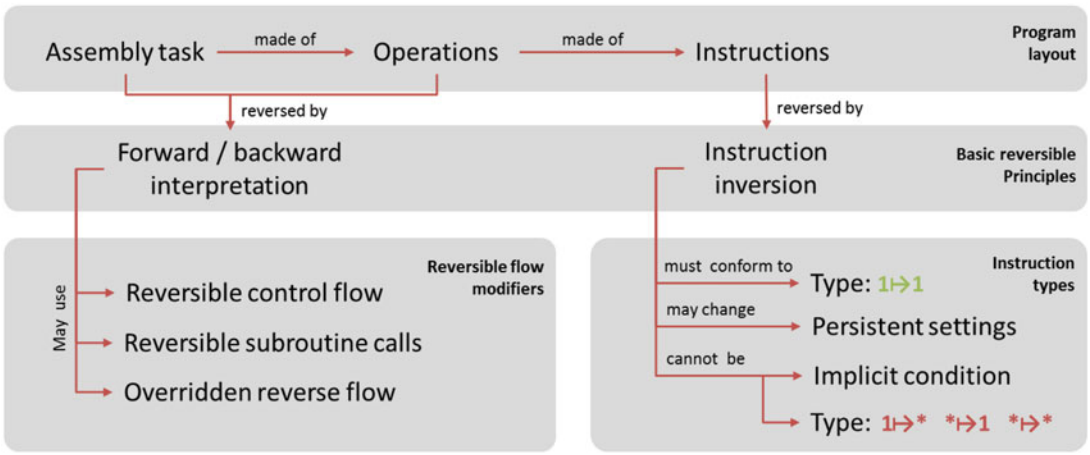


Fig. 5. Principles of reversible assembly programming.

A directed-cyclic graph is used to model the underlying reversible assembly sequence. In this graph, each node corresponds to a primitive instruction that is executable on the physical platform. Furthermore, each node contains pointers to the next forward instruction and the next reverse instruction. Overall, the graph is evaluated through forward/backwards interpretation and each instruction is evaluated using instruction inversion with different semantics for forward and backwards execution. Figure 6 shows an example of such a graph. This figure represents part of the graph actually generated when programming one of our test cases. The graph-based approach allows modelling of programs with directly reversible sequences (fully connected in both ways), indirectly reversible sequences (different forwards and backwards arcs), and nonreversible sequences (by marking nodes as nonreversible and only having as single connection).

Characteristics of the execution model are as follows:

- The default and directly reversible model is a linear graph without branching, relying on the principles of forward/backwards interpretation and instruction inversion.
- The model uses nested branches for specifying alternative reversible flow. This enables modelling of indirect reversibility at the operation level.
- The graph does not allow either reversible or nonreversible runtime control flow structures apart from those used for changing the execution direction. Control flow structures can, however, be used in the implementation of individual instructions.
- The execution model does not include sequence composition using procedure calls. The concepts of operations and reversible subroutines are handled in the programming language. Operations and subroutines are concatenated together at compile time before instantiation in the model. While this would not be a feasible approach in general-purpose programs, this is done automatically and works for small programs such as assembly sequences, and significantly simplifies the implementation of reverse execution.

A UML diagram of the model can be seen in Fig. 7. The execution model has primitive instructions that are used for common interaction with robot and gripper, as well as implementation of I/O, waiting, and delays. It also contains commands for performing error detection checks based on I/O values and robot positions. Finally, the model allows users to integrate their own instructions for more complicated robot behaviour. Our instruction set resembles a condensed version of the instruction sets found in many traditional robot scripting languages, which is convenient as the instructions will be familiar and are closely linked to the physical actions on the platform. Each instruction includes a set of flags that specify additional details as to how the instruction acts when executed forward or backwards: The flag *is_reversible* marks whether the instruction is reversible, and *is_swapped* switches the forward and reverse semantics internally in the instruction, such that it acts opposite to the execution direction. The use of these properties is explained in the rest of this section.

https://doi.org/10.1017/S0263574717000613 Published online by Cambridge University Press

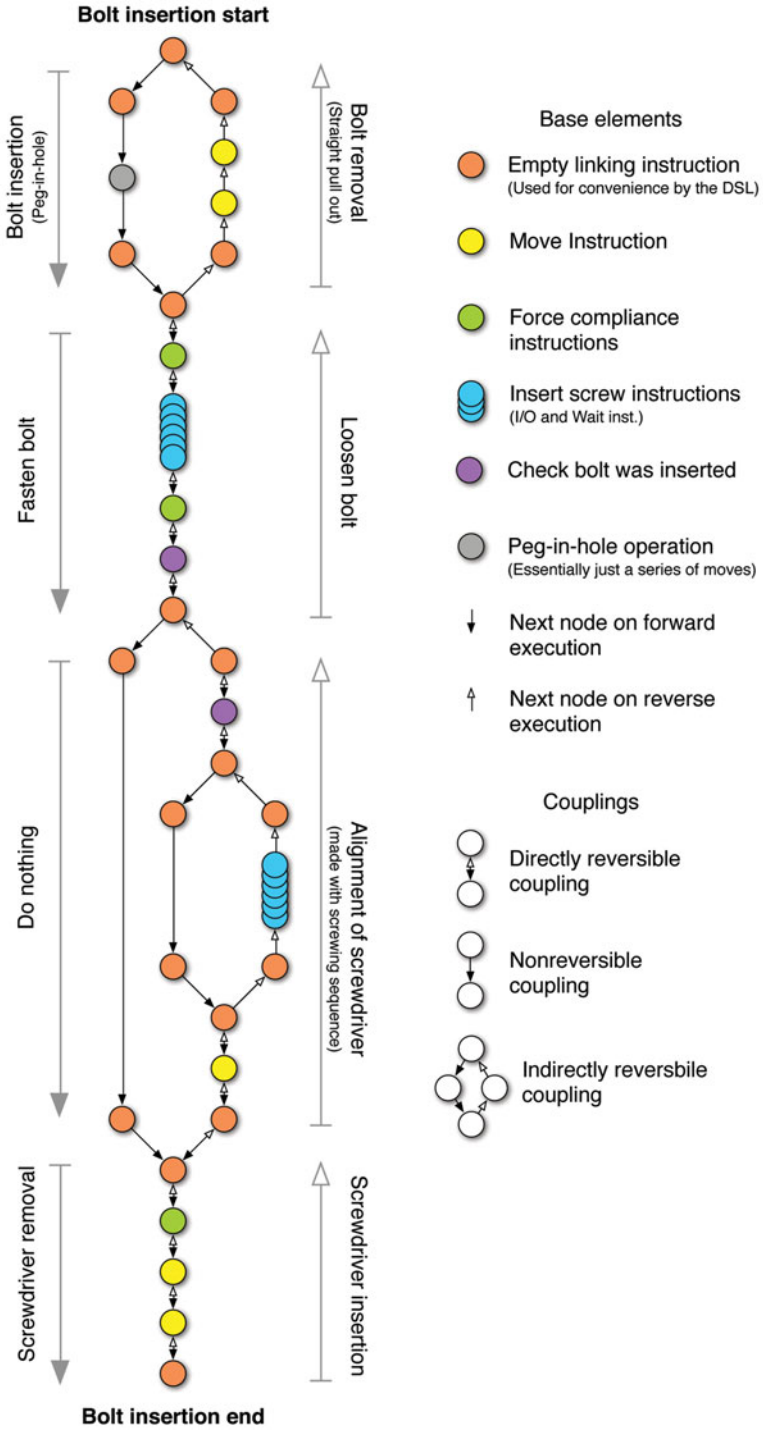


Fig. 6. The execution model for an operation placing a bolt in a bracket, shown as a cyclic-directed graph. In this graph, reversible nodes have pointers to the next instruction for both forward and reverse execution. Nonreversible nodes are only connected in a single direction.

5.2. Model characteristics by example

Figure 6 shows the execution graph for an actual operation that inserts a bolt into a bracket. On reverse execution, the screwdriver needs to align with the bolt before it can actually unscrew and remove the bolt. This alignment is done by performing a screwing operation.

This assembly operation contains a mix of directly and indirectly reversible steps, which makes the graph fairly complex. This complexity is not a representative of the program as a whole; the vast

https://doi.org/10.1017/S0263574717000613 Published online by Cambridge University Press

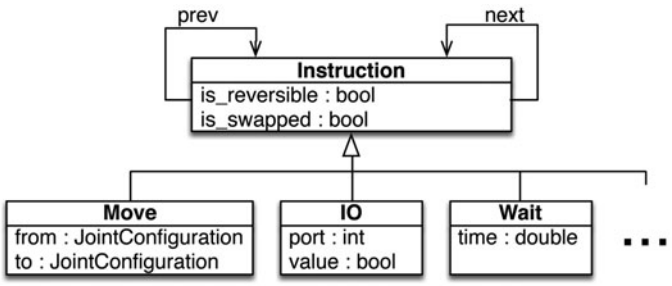


Fig. 7. UML diagram of the execution model.

majority of the program relies on the default reverse constructs with directly reversible sequences (see Section 5.3). Nonetheless, the graph is useful for illustrating the different principles and possibilities of our framework, as discussed below.

5.2.1. Changing the default reverse behaviour. The default reverse behaviour is changed in the part of the graph marked *Bolt insertion*. The *Bolt insertion* is indirectly reversible, and thus uses the principle of overridden reverse flow. We use a especially designed peg-in-hole action for inserting the bolt at a skewed angle, but as an inverse the bolt is simply removed with a single straight-line motion. The graph has several cycles, caused by the indirectly reversible operations, where different forward and reverse execution paths exist.

5.2.2. Forward/Reverse-only behaviours. The part of the graph tagged, *Alignment of screwdriver,* shows an example of reverse-only behaviour. By overriding the default reverse of an empty operation using indirect reverse behaviour, instructions can be used in a forward/reverse-only behaviour. Inserting the screwdriver into the slot of the bolt is very error-prone. This is due to the uncontrolled rotational alignment between the bolt and the tip of the screwdriver. If the two do not connect the bolt is not unscrewed correctly. An easy way to ensure that screwdriver and bolt are connected before starting the unscrewing process is to start with a screwing operation that inserts the bolt (even though the bolt is already inserted). This jiggles the screwdriver slightly, causing it to align properly. This operation only needs to be executed during reverse execution where the bolt is removed.

5.2.3. Nonreversible instructions. Nonreversible instructions are found in the reverse execution sequence where the graph is labelled *Alignment of screwdriver*. We observe that although a single instruction in a line of instructions is nonreversible (they have no forward pointers), using manual override of reverse behaviours, a bidirectional transformation can still be established at an operational level. Real assembly cases include a mix of reversible operations (e.g., picking, placing, and screwing) and inherently nonreversible operations (e.g., gluing objects together and drilling holes). To distinguish these cases, all instructions are tagged through the variable *is_reversible*.

5.2.4. Swapped semantics. The *is_swapped* variable is used when creating overridden reverse flow. This is exemplified by the part of the graph in Fig. 6 tagged *Alignment of screwdriver*. The *is_swapped* variable signals an instruction to switch its forward and reverse semantics, thereby flipping direction in the *one-to-one* property of instructions. The *is_swapped* variable is used to integrate the principles of instruction inversion and overridden reverse flow. When stepping through the model in a forward direction instructions are evaluated using their forward semantics and vice versa when executed in reverse. Instruction arguments are always provided in terms of the sequence execution direction (forward or reverse); hence, without the *is_swapped* there would be no means of internally switching, and a user would need to be extra careful when implementing instructions specifically intended for overridden reverse flow.

5.2.5. Forward versus reverse sensing. An instruction is used to check if a screwing operation was successful. It only works in the forward direction. In general, sensors cannot be assumed to work in both directions. A sensor that can detect an error forwards can for instance not necessarily be used for detecting the same type of error when execution is reversed. The screwdriver uses its internal torque sensor to determine if a bolt was correctly inserted (if not an I/O signals the framework that an error occurred). The screwdriver is, however, not capable of determining whether the bolt was correctly

https://doi.org/10.1017/S0263574717000613 Published online by Cambridge University Press

```
n := initial node
d := initial direction
e := initial error information
while(n ≠ null ∧ ¬stuck(e))
  d' := { d,                          if n.is_swapped == false
        { reverse direction of d,     otherwise
  r := evaluate(n.instruction, d', e)
  if( r = continue )
    e := update_err_state(e,n)
    n := next(d,n)
  else
    d := reverse direction of d
    e := insert_error_info(d,e,r,n)
```

Fig. 8. Execution algorithm with generic error handling.

removed when the program is executed backwards. Implementing error checks in a consistent and correct manner is the responsibility of the user. Users determine in which direction-specific checks are performed. This and the concept of error detection are described in more detail in Section 7.2.

5.3. Directly versus indirectly reversible

It is our intention that assembly programs rely on the principles of program inversion (achieved through directly reversible sequences and the *one-to-one* pattern) for inverting the majority of programs. As noted earlier, the graph in Fig. 6 can give the impression that the majority of a program is inverted on an operational level using indirect reversible sequences. This is however not the case: The sequence in Fig. 6 is chosen for its complexity and for the specific purpose of showing the possibilities of overriding the default reverse code.

One of the strong arguments for using program inversion for reversibility is that it enables the program interpreter to change the execution direction at any point during the program. Overriding the default mechanism will sometimes break this principle. Indirectly reversible operations may, however, themselves be implemented such that they can be reversed half-way through.

A distinction is made between atomic and nonblocking implementations. Atomic implementations restrict direction change until the entire sequence of instructions has been executed. This is, for instance, the case in the part of the graph marked *grasping of screw*. Nonblocking implementations are able to change direction midway and are therefore preferred. Sequences such as pushing an object may only make sense in the forward-only graph, but can be executed backwards.

5.4. Execution algorithm with error handling

A key advantage of the directed graph model is the simplicity of the reversible evaluation algorithm, shown in Fig. 8. The program is executed from a given node *n* (normally the beginning or the end of the assembly sequence), in a given execution direction *d* (forwards or backwards), using error control information *e* (explained below). The algorithm is bidirectional (works for execution in either direction) and is generic in terms of error handling (under what conditions backtracking is done and how far).

Execution proceeds until there either is no next node to execute or error handling has become stuck (line 4). The execution, thus, only stops if the beginning or end of the program is reached (no next node), if it encounters a nonreversible instruction during reverse execution (no next node), or if the error control information *e* indicates that execution has become stuck.

To properly handle the `is_swapped` node property (described in Section 5.2.4), an argument direction d' is calculated (line 5). d' is computed based on the current direction d and the direction in which the instruction is designated to execute. It is basically a simple way of making instructions in reverse branches execute their forward behaviour. The instruction of a node is evaluated using the function `evaluate` (line 6). The instruction is evaluated in the direction d'. This determines the semantics used for the evaluation; an I/O could be turned on during evaluation in a forward direction and off during backwards. The function `evaluate` ensures that no constraints in the error control state *e* are violated. Depending on the outcome, the `evaluate` function returns information on how to proceed.

https://doi.org/10.1017/S0263574717000613 Published online by Cambridge University Press

This information is stored in the variable r, and is either `continue`, indicating that no errors occurred, or information about what error occurred (line 7).

If no errors occurred (lines 8 and 9), the error state is updated (see below) and the next node is selected for evaluation. The node is selected in accordance with the direction and the `is_reversible` property. Conversely, if an error occurred (lines 10 and 11), the direction is reversed and the error state is updated to reflect the new error requirements.

The algorithm as presented here is generic in that it allows us to express several different approaches for controlling errors. The error control information e essentially represents the state of the execution with regards to error handling, and can be used in different ways depending on the error handling procedure. As a simple example, e can be a counter that is incremented once for every step, and the `stuck` function can impose a maximum value `MAX` on this counter (e.g., twice the length of the assembly sequence), as follows:

$$\texttt{initial error information} \equiv 0$$
$$\texttt{stuck}(e) \equiv e < \texttt{MAX}$$
$$\texttt{update_err_state}(e,n) \equiv e + 1$$
$$\texttt{insert_error_info}(d,e,r,n) \equiv e + 1$$

Alternatively, and probably more interestingly, we can associate a counter with every node, that is incremented if an error is triggered while executing this node. The `stuck` function can here impose a per-node maximum value:

$$\texttt{initial error information} \equiv \{(n,0) | n \texttt{ node in program graph}\}$$
$$\texttt{stuck}(e) \equiv \exists (n,i) \in e : i > \texttt{MAX}$$
$$\texttt{update_err_state}(e,n) \equiv e$$
$$\texttt{insert_error_info}(d,e,r,n) \equiv \{(n,i+1)\} \cup E, e = \{(n,i)\} \cup E$$

More elaborated schemes such as a region-based try construct[10] can also be supported. Section 7 goes into more detail about how errors are detected and handled.

Using reversibility, as an error-handling mechanism imposes the somewhat unusual requirement that programs must be able to change execution direction at any time and at unforeseen places during run-time execution. Our execution algorithm relies on reverse interpretation for achieving this. This is, in contrast, to full program inversion, which is more suited for programs that execute from start to the end without changing direction.

5.5. Compile-time state propagation

The tasks of specifying instructions with correct parameters and populating the model in a consistent and correct manner are simplified by running an analysis of the model instance before execution. The analysis has the dual purpose of (1) correcting errors related to instructions used as *implicit conditions*, and (2) reducing the syntax required for specifying instructions where reverse arguments can be deduced (*many-to-one*).

The problem in case 1 is that instructions that are used as *implicit conditions* give incorrect behaviour when backtracking. A sequence that starts with closing the gripper, does something, and then closes the gripper again will result in incorrect behaviour if reversed. The second close gripper instruction is redundant and the instruction is therefore removed in the analysis.

The problem with case 2 is that due to the design of hardware interfaces and robot scripting languages instructions often follow the *many-to-one* pattern. This is sufficient for programming forwards sequences but gives problems when sequences are executed backwards. The execution model relies on instructions following the *one-to-one* pattern. The move command, therefore, encodes a relative change in joint configurations rather than an absolute final destination. This relative positional change is specified using both *from* and *to* configurations as the instruction arguments. This means that in commands tha move the robot from position A to B and then C, the mid-point B is both the destination of the first move and the starting position of the second. B is both the *to* in move AB and the *from* for move BC. By providing this analysis, instructions work in a reversible context by following the *one-to-one* pattern, while still allowing users to specify instructions in a familiar syntax with absolute position encodings.

The analysis works by having the system state transitively passed around during an initialization process. Here, each instruction receives the state, records it, and optionally changes it, before passing

https://doi.org/10.1017/S0263574717000613 Published online by Cambridge University Press

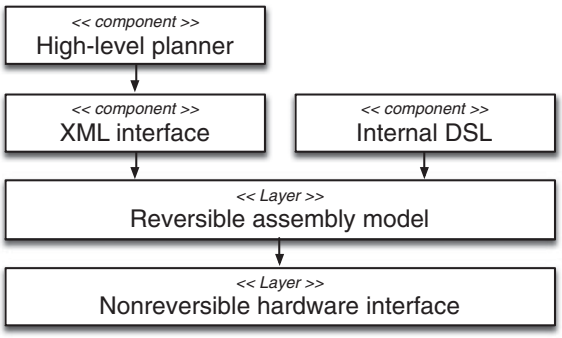


Fig. 9. The overall framework architecture along with connections to hardware, high-level planners, and users.

it to the next instruction in the forward and backwards directions. Instructions are thereby passed their *from* state—which they may record, as in the case of the move example; or check if their instruction is redundant, as is the case with the gripper example. By passing the system state around, we also ensure that different types of instructions that change the same physical property (e.g., robot position) use and modify the same state variable. This is, for instance, the case with move, force compliance, and action instructions. Note that *one-to-many* instructions such as the force compliance command put the robot in an potential unknown state— the model has no way of knowing where the robot is afterwards. Due to the lack of a better alternative, a move command that receives an unknown state simply puts the robot into its *to* configuration when executed backwards.

6. Programming Language

We provide both a domain-specific language (DSL) and an XML-based representation to instantiate the graph-based execution models from concise descriptions of assembly sequences. The XML interface populates the execution model directly and allows the framework to be integrated with other software components. Through the DSL, users can directly program assembly tasks into the framework. The programming language and the execution model share the responsibility for creating reversible programs. The execution model provides the flexibility for representing the reversible assembly sequence, and the language populates the model in a consistent manner. Figure 9 shows the overall architecture of the software. Figure 3 shows the general programming model used.

The following section explains the DSL and outlines the interesting features and possibilities arising from programming in a reversible language. The XML syntax employs a fairly direct and less interesting XML-representation; hence, the approach is only mentioned briefly.

6.1. Language design

The DSL is implemented as an internal C++ DSL employing a standard method chaining approach.[23] The DSL is named SCP-RASQ (Simple C++ Reversible Assembly SeQuences), and is simple as far as language features are concerned, but it is operational and thus serves to demonstrate how a reversible robot language can work. A more complete version of how we envision a fully-fledged reversible assembly language has been proposed in,[10] the operations of this language are however only theoretical. Connecting the advanced language proposed there with the framework described in this paper is future work.

SCP-RASQ is based on an existing DSL for describing assembly sequences,[7] and has been styled as a traditional robot scripting language but with added features for reversibility. Using SCP-RASQ, individual instructions are arranged and organised into sequences. Sequences can, in turn, be invoked from other sequences, much like procedural calls in general-purpose programming languages. Sequences can also be *uncalled* thereby invoking the reverse behaviour. This is discussed in Section 6.2. Apart from nested sequence calls, the instructions closely resemble those of the execution model. These parts of the language instruction set enable the programmer to express forward executable assembly programs.

Our intention is that the additional information needed for reverse execution is kept minimalist and unobtrusive. As we have previously argued, physical constraints and real-world limitations may

https://doi.org/10.1017/S0263574717000613 Published online by Cambridge University Press

$$
\begin{array}{lcl}
P & := & (\ IO \mid JointConf \mid Condition \mid Sequence\ )* \\
IO & := & \underline{\text{IOPorts}}\ N_{\mathsf{io}}\ n \\
JointConf & := & \underline{\text{JointConfiguration}}\ N_{\mathsf{q}}\ vector \\
Condition & := & \underline{\text{Condition}}\ type\ N_{\mathsf{con}}\ args \\
Sequence & := & \underline{\text{sequence}}\ N_{\mathsf{seq}}\ RevFlow\ Operation* \\
Operation & := & (Instruction \mid NestedCall)\ RevFlow \\
Instruction & := & \underline{\text{move}}\ N_{\mathsf{q}} \mid \underline{\text{io}}\ N_{\mathsf{io}}\ n \mid \underline{\text{check}}\ N_{\mathsf{con}} \mid \ldots \\
NestedFlow & := & \underline{\text{call}}\ N_{\mathsf{seq}} \mid \underline{\text{uncall}}\ N_{\mathsf{seq}} \\
RevFlow & := & (\underline{\text{reverseWith}}\ N_{\mathsf{seq}})?\ \underline{\text{nonreversible}}?
\end{array}
$$

types and *args* specifies conditions and arguments,
$N \in$ names, $n \in \mathbb{R}$, *vector*$\in \mathbb{R}^n$

Fig. 10. BNF of the abstract syntax for the internal DSL.

necessitate specifying and overriding the default reverse approach. This is done through the creation of indirect reverse sequences, explicitly specified by the user. We provide language constructs enabling overriding the default reverse with alternative code for either entire sequences, specific instances of the sequences, or just single commands. We also allow the user to explicitly specify the reverse behaviour at a given program point.

6.2. Calling and uncalling sequences of operations

Tetsuo *et al.*[9] describe the possibility of having both procedural `call` and `uncall` instructions, which enables a sequence to be invoked for both forward and backwards execution. Transferring this concept to assembly programs can reduce the programming task and the amount of source code that has to be maintained. For example, a place function could be derived from a pick function, or two complicated move paths from A to B and B to A could be contained within a single sequence of instructions. In our concrete cases, we, for instance, use this functionality with a *pick screwdriver* operation, which is invoked using `uncall` when returning the screwdriver after use.

6.3. SCP-RASQ syntax

The Backus-Naur Form (BNF) of the abstract syntax of SCP-RASQ is shown in Fig. 10 (omitting syntactic noise caused by the use of an internal DSL). An example of a program is given in Fig. 11.

Declarative parts consist of joint configurations (*JointConf*), I/O ports (*IO*), and different conditions (*Condition*) used to check for error and to determine when external equipment is ready.

Declarations are used to specify sequences (*Sequence*), which are lists of operations (*Operation*). These operations can be primitive instructions (*Instruction*) such as move commands, I/O operations, close gripper, wait a fixed time, wait for a condition, and print to screen, but may also include more advanced instructions, such as the force instructions that make the robot apply force in a given direction, and the action instruction that activates operations from an underlying library. In our case, actions are pre-programmed movements that are automatically parameterized using simulation.[7] Our actions are essentially just a compilation of pre-programmed move commands. Finally, check instructions are provided with a condition as argument that if evaluated to false, signals the controller that an error occurred.

Operations in sequences can also be used for invoking other sequences (*NestedFlow*). Sequences cannot be called recursively and as the execution model is a static model there are no flow operators (loops and conditionals).

Instructions specifying nonreversible and indirect sequences (*RevFlow*) can be applied to entire sequences, specific instances of sequences, or individual instructions. The command `reverseWith` specifies an alternative sequence to be executed in the reverse direction and the command `nonreversible` tags operations or sequences as nonreversible. We note that static loops and conditionals (e.g., for loops with fixed bounds and configuration-dependent conditions) can be used in SCP-RASQ using normal C++ syntax, as SCP-RASQ is an internal DSL embedded in C++ with an underlying semantic model. Loops and conditions execute using C++ semantics during the creation of the underlying execution model, which, for example, allows them to be used to build extensive graphs describing repetitive and near-identical tasks.

The example in Fig. 11 shows part of the code used for generating the graph on Fig. 6. The code shows how three sequences are created, which are all used for inserting screws into a brackets placed

https://doi.org/10.1017/S0263574717000613 Published online by Cambridge University Press

```
// DECLARATIONS
  JointConfiguration qScrewOutside( ... );
  JointConfiguration qScrewInside( ... );
  IOPorts screwdriver(7);
  IOPorts screwdriverBackwards(6);
  IoComparison screwingFinished ( IOPorts(5), Switch::on );
  IoComparisonMonitored screwingSucceded ( ... );
  ...
// SCREWING SUB-OPERATIONS
  sequence("screwdriver_activate").
    io(screwdriver, Switch::on).
    wait(0.3).
    wait(screwingFinished).
      reverseWith("screwing_finished_backwards");
    io(screwdriver, Switch::off).
    io(screwdriverBackwards, Switch::off);

  sequence("insert_screw_suboperation").
    check_start( screwingSucceded ).
    forceMode( pushRight , deativate).
    call("screwdriver_activate").
    forceMode( deativate , pushLeft).
    check_stop( screwingSucceded ).
    print("empty_ins").reverseWith("screw_grasp").
    forceMode( deativate , pushRight);
  ...
// SCREWING OPERATION
  sequence("insert_screw_operation").
    action(insert_screw).
        reverseWith("remove_screw").nonreversible().
    call("insert_screw_suboperation").
    move(qScrewInside).
    move(qScrewOutside);
```

Fig. 11. Assembly sequence in SCP-RASQ. The sequence includes information on how a screwdriver is activated, how to check if the screwing succeeded, what force compliance commands to activate, and finally, how to move the robot to and from the screwing location.

on a fixture. Through combinations of nonreversible functions and indirectly reversible flow, we can build abstractions of indirectly reversible behaviour, as exemplified in Section 5.2. By applying both nonreversible and indirectly reversible flows, mutually exclusive instruction sequences are created, which restrict direction changes partway through the execution.

6.4. Transforming SCP-RASQ into the execution model

The execution model is populated based on information extracted from SCP-RASQ programs. In order to reduce the complexity of the conversion, we have split the process into two phases: First, all nodes in the execution model are created, and second they are linked together. The conversion process is based on the following principles:

- A sequence starts and ends with an empty node, this makes linking of indirect reverse sequences easier.
- A sequence can be built in both forward and reverse direction. The direction is passed on to its nested sequences and embedded in instruction nodes in the execution model using the variable *is_swapped*.
- Each call and nesting of a sequence results in the construction of a new set of nodes in the execution model.
- All nodes are fitted with an ID tag such that the actual linking can be done at a later stage.
- A sequence provides a special ID tag. This lets nodes, which nest or reverse the sequence, relate to sequence start and end elements. This is contrary to normal instruction elements, which do not differentiate between start and end handles.
- The linking of nodes is done recursively from the ID tags after all nodes have been constructed.

https://doi.org/10.1017/S0263574717000613 Published online by Cambridge University Press

6.5. Assessment of SCP-RASQ

Prior to the work presented in this paper, we implemented two working examples of a DSL for assembly tasks. The first implementation used our nonreversible DSL,[7] which closely corresponds to the nonreversible part of SCP-RASQ. We then went on to extend this framework into a reversible language.[11] We now present a more refined version where the underlying reversible framework has new and more robust features, enabling improved experimental results to be demonstrated. Moreover, the uncall instruction has been implemented.

These previous assembly languages were used to implement the same use case; hence, we are able to do a comparison of these programs and assess the work required to convert a program into a reversible program. Comparing the three programs, while ignoring unimportant differences (comments, whitespace, etc.), we find that significant parts of the code remain unchanged. The original forward-only program was implemented in 525 lines of code and the first version of the reversible program was implemented in 575 lines of code. The version presented here, with improved features, was implemented in 500 lines of code. The initial increase in lines of code came mainly from adding check conditions and from a necessary refactoring of a complex screwing operation. This newest reduction in lines can mainly be contributed to the uncall feature. The addition of the uncall feature also means the program code is more versatile, as all operations can be used backwards. Even if these are not used backwards in the final program, they are often convenient during debugging and the creation of programs.

We, do, however, observe that when implementing complicated indirectly reversible behaviour, such as the screwing operation shown in Fig. 6, the source code also tends to become complicated. The addition of the uncall feature has not changed that. Better syntax and higher-level language for specifying these complex situations are needed and will be developed in future work. The uncall feature does, however, motivate proper creation of code where direction changes can occur at almost any point in the program, rather than being limited to after the execution of an entire sequences.

6.6. XML language

Our XML-based interface is built and implemented on the same basic infrastructure as the DSL. Through this interface, assembly programs can be created following the same programming style as used for the DSL and shown previously in Fig. 3. The XML interface makes it easy to integrate planners, languages, and other tools with the framework. The main advantage of using the XML representation in comparison to the internal DSL is that the XML interface integrates dynamically with the execution model. This means that the entire framework does not have to be recompiled every time the assembly program is changed.

7. Error Handling

Our main application of the reversible assembly programming language is as an error handling mechanism. With a system capable of backtracking assembly sequences and repeating the assembly processes, many errors can be rectified without the need of implementing custom error handling.

7.1. Error types and recovery capabilities

Reversible execution and backtracking can be used as the default error handler, but in some circumstances it is insufficient. We distinguish between *major* and *minor* errors. Reversible execution is useful for minor errors, that is, errors that prohibit the assembly from continuing correctly but that do not cause a significant change in the setup. Conversely, major errors cannot simply be solved by backtracking and repeating, e.g., hardware failures or errors in nonrecoverable processes.

In our framework, error-conditions are specified by the user and errors are triggered by differences between the expected and actual state of the workcell. Examples could be expected I/O values, joint configurations, force limits, and so forth. For these cases, the system will attempt automated error recovery through reverse execution.

In some error cases, the most efficient error handling strategy is to discard the workpiece and restart the process, e.g., if the workpiece is defective or trying to undo a click-fit that may break the

https://doi.org/10.1017/S0263574717000613 Published online by Cambridge University Press

workpiece. Although not the intended purpose of indirect reversible behaviour, we could also use it to implement routines that discard a workpiece and repeats the assembly.

7.2. Reversible error detection

One of the benefits of using a reversible execution model for handling errors is that the execution direction can be changed anywhere in the program, meaning that backtracking and retrying can act as the default error handler anywhere in the program. Integration of an automated error detection process is considered outside the scope of this paper but could be used to take further advantage of this property. The errors that are addressed are based on pose-uncertainties or variations in sensor readings. This makes the errors difficult to detect directly, and we often rely on indirect detection of failures, e.g., deducing if two brackets were aligned correctly based on if a bolt could be inserted into the two. However, as our solution for correcting the error is the same regardless of cause the exact origin of the error is less important.

7.2.1. Approach. Error condition checks are implemented as instructions in the execution model, which signals the program-interpreter if an error occurs. Furthermore, users can specify if conditions are evaluated in a single direction, similar in either direction, or are two different conditions for forward and reverse directions.

Currently, all checks are based on manual input of expectations, which are specified using the following check types:

Point evaluations: During the execution of the check instruction, the expected state of the workcell is compared against the actual state.

Delimited evaluation: Evaluations like a distance measurement require both a start and an end instruction. The measurement is started at a given point, and evaluation occurs at a later point.

Through these instructions, we can test various conditions related to our use-cases. Communication with the torque sensor in our screwdriver is, for instance, done through I/O, and robot position can be compared to either absolute or relative positions.

7.2.2. Error detection in a reversible context. In a reversible system, we would ideally like to have direction-invariant error detection instructions. If these instructions are direction-invariant, then a condition that works as a pre-condition during forward execution becomes a post-condition during backwards execution and vice versa.

Direction-invariant checks are, however, not always possible and working with physical systems means some things cannot be tested during reverse execution. Checks related to the state of the workcell, such as *screw is in object*, works for both forward and reverse execution. Checks related to an action having been performed, such as *screw was inserted* do not. Making error checks that are direction-invariant is not always possible due to indirect errors and forward-only sensing. In a screwing operation, a torque sensor can be used to measure if a screw was correctly inserted but not removed.

In our model, we support both direction-invariant and noninvariant conditions, but users need to be aware of the difference. For a noninvariant check, such as *screw was inserted*, the intuitive reverse test is *screw was removed*. Implementation-wise, they are however, two different instructions in the execution model and not each other's inverse. For a sequence of instructions that activates the screwdriver and inserts a screw, they are the pre- and post-condition for forward execution. This is illustrated in Fig. 12 where a screwing example is visualised.

Our implementation does not allow error checks to change the physical state of the robot workcell. This ensures consistent execution behaviour that is independent of how many times and in which direction checks have previously been executed.

7.3. Error correction: design and implementation

We have argued for the use of reverse execution as our default approach to error correction. Given the model for reversible execution, there are still a few design and implementation choices remaining that are outlined in this section.

https://doi.org/10.1017/S0263574717000613 Published online by Cambridge University Press

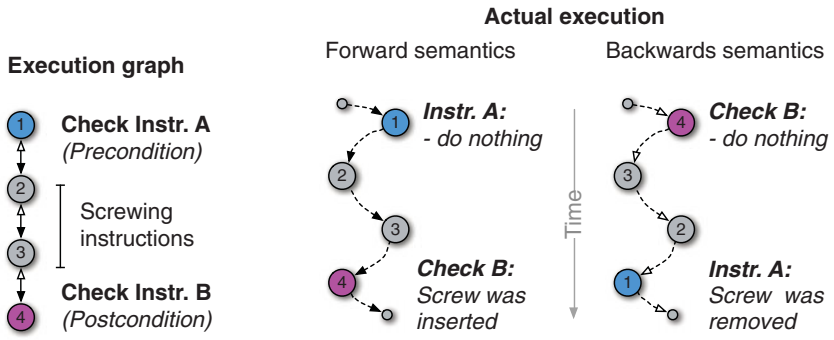


Fig. 12. A screwing sequence checks for the correctness of the screwing process and screw placement.

7.3.1. Errors in errors. Our current approach to error handling is to switch the execution direction. We also apply this approach to errors occurring during the recovery process, hence the execution direction is then returned to normal when an error occurs within an error.

If an error check fails, instructions for undoing the operation that led to the error are executed. This is true no matter how many errors occur and it is the user's responsibility to make the necessary checks. Our error handler has a default strategy for handling errors anywhere in the program and is capable of handling errors within errors. These are normally properties associated with high-level symbolic planners where replanning occurs after an error. Our approach is more dynamic than the constructs used in knowledge-based error handling systems, try-catch constructs, and robot languages error handling routines.

Our system targets errors occurring as a result of stochastic effects. As a result, errors in errors are not necessarily unrealistic. This is also why we address these with a switch in execution direction rather than stopping the system and waiting for manual restart.

7.3.2. Defining error checks. Ensuring program correctness comes down to how error checks are defined. In general, there are three scenarios that users need to account for:

- Error checks are conceptually correct when executed both forward and backwards.
- A forward error check may lead to termination at program start and vice versa.
- Double flip of execution direction must not lead to error checks being bypassed due to indirectly reversible sequences.

Moreover, if two consecutive errors occur within an assembly, the first error implies a failure to put together the object. The second error implies that the system was unsuccessful in disassembling the object again. If the error is minor, the object can be assumed to be assembled, and normal execution can be resumed. If the object was not correctly assembled, this should be detected during the first check or in another error check. If the program is consistent, these errors within errors are caught before the program reaches termination.

Consider the case where we have to move a nut from one bolt to another. If the insertion onto the second bolt fails, the program backtracks and tries to reinsert the nut onto the first. This behaviour can loop between the two insertions until one case succeeds or the program is terminated due to a time-out criteria. In the case that error detection is not made consistently across both execution directions, it is however possible that the program will reach termination in the opposite direction of the error check. For instance, in an assembly task where a screw is inserted and the system is only capable of detecting whether the screw was inserted successfully: If the screw was somehow dropped during the assembly sequence, then the program might eventually backtrack to its beginning—leaving the error unsolved but also the program uncompleted.

In the case that the system enters a loop between two errors, we provide a step counter as guarantee of termination. Error detection instructions should, however, be implemented such that the system does not enter busy waiting, that is a consecutive error that keeps spinning the execution direction back and forth without the system doing any work. For this not to occur, error detection must allow the robot to cycle back and forth in the program. For instance, if a screw is dropped during forward execution, the dropped screw cannot constitute an error in the backwards direction. Here, the

https://doi.org/10.1017/S0263574717000613 Published online by Cambridge University Press

robot needs to be allowed to backtrack and repick a new screw. Using our pragmatic solution with indirect error check instructions, avoiding busy waiting is a simple matter of the user implementing sensible error detection instructions and choosing where to check conditions. In automatic detection systems, more advanced strategies for what constitutes errors would be needed to avoid busy waiting.

7.3.3. How far to reverse. Our current implementation uses a deterministic approach, where the number of instructions to reverse is gradually increased each time the same instruction throws an error. The first time the system encounters an error it backtracks a specified number of steps before it resumes and repeats the operations. If the system then encounters the same error again it backtracks further before resuming once again. This continues until the error is solved or the system cannot reverse any further.

We have previously conducted experiments with random strategies[11] and also proposed the idea of having users place hints and labels in the program to specify return points.[10] We discarded the random approach as it was very difficult determine an efficient number of steps, due to variation in the number of instructions required for each conceptual operation. Overall, we consider the following trade-offs:

Blind versus supervised strategies are about using sensors to determine how far to reverse. When encountering an error, the system should reverse until it corrects the uncertainties causing the errors. Our tested approach searches for this point blindly, but active strategies based on sensor and vision systems could be applied. Our blind strategy is general and fairly easy to setup, which makes it suitable for small-batch production. Sensor-based strategies could potentially ensure more efficient search.

General versus specialized strategies involve searching either entire programs or backtracking to a explicitly specified target. In specialized strategies, human intuition can be used to explicitly describe backtracking points for efficient handling of specific and commonly occurring errors. In this paper, we prefer a general approach as it provides a good proof of concept, in commercial systems a mix would probably be preferred.

Deterministic versus learning concerns strategies that apply learning. If the same instruction throws an error in two consecutive assemblies of the same product, the current strategy applies the same naive search in both cases. This is of course a very simple and deterministic approach but learning could be applied to create faster searching strategies.

8. Experiments

Error recovery using reverse execution is tested using the two industrial assembly tasks shown in Fig. 13. Common methodology for the experiments is described in Section 8.1 and the use cases are described in Sections 8.2. Sections 8.3 and 8.4 provide details on the experiments performed.

8.1. Methodology

To test our concept of reversibility, we use our physical robot platform along with the developed framework. Both the platform and the methodology surrounding the framework is described here.

8.1.1. Physical setup. The platform depicted in Fig. 2 was used to test the software and solutions developed. All experiments were conducted on a physical robot platform. The platform comprises: a Universal Robot UR5 arm, a Robotiq 2-Finger 85 gripper, and a Desoutter CVIR-II/HP2 75J screwdriver. The framework only utilizes the sensors found in the screwdriver, gripper, and robot manipulator. The Desoutter screwdriver is an industrial screwdriver normally used by humans. Rather, then using an expensive robotic screwdriver a small mounting has been created, which allows the screwdriver to be picked using the Robotiq gripper. The screwdriver can be returned and picked from a pneumatic operated stand. On the platform modularised pallets can be mounted. Each pallet is equipped with a single connection of air, Ethernet and power. The pallets have their own pneumatic circuits and a MOXA ioLogik E1213 board for controlling I/O. This means pallets can be swapped for different tasks.

8.1.2. Software setup. An SCP-RASQ program for each of the two use-cases is created. The programs follow the logic flow described in Section 8.2. The Robotiq gripper is integrated into the instruction set of the framework, while the screwdriver and pneumatic components are interfaced through I/O

https://doi.org/10.1017/S0263574717000613 Published online by Cambridge University Press

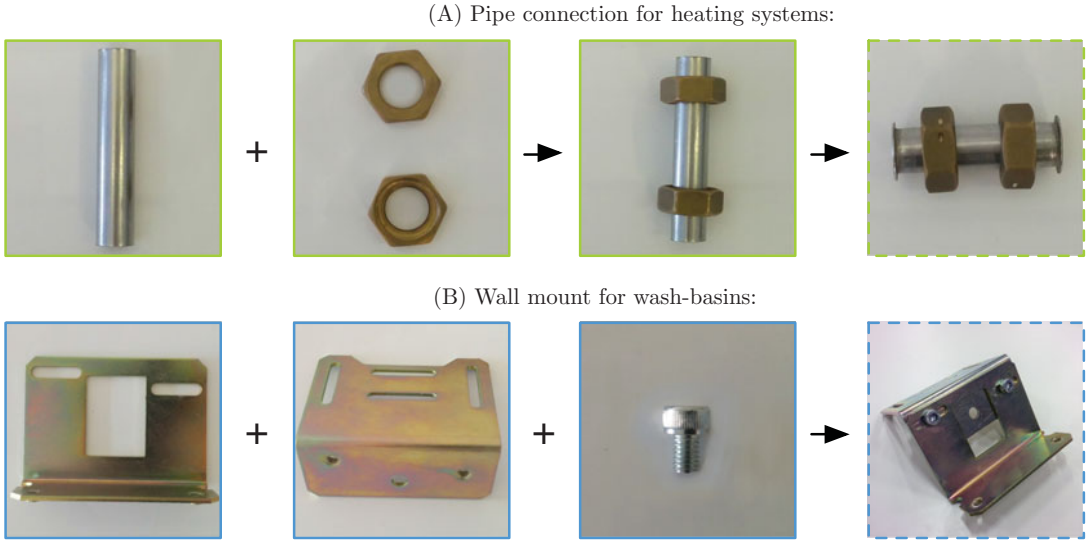


Fig. 13. Industrial assembly cases used to demonstrate the system in practice. (A) Pipe connection for heating systems. (B) Wall mount for wash-basins.

commands. The robot is mostly moved using point-to-point movements except when union nuts are picked (case A) or the screwdriver is active (case B). In these cases, the robot uses its force compliance to push in the required direction.

Both cases include a final step where the finished product is discarded into a box. This step was not performed when running the programs backwards, as it is a nonreversible task since our current setup cannot bin-pick the part out again. We consider this acceptable as the purpose of the test is not to identify conceptual nonreversible operations, but to identify problems with reversible operations that are difficult or nonreversible when preformed with robots.

Setting up automated assembly takes a long time and involves many different tasks, from designing feeders and grippers to the tuning and tweaking of process parameters and robot configurations. Lab experiments can, therefore, easily be tuned to exclude different sorts of errors or increasing other types of errors. Similar to the approach used by Neto *et al.*,[5] we have attempted to be true to our idea of using automation in small-batch productions with the experimental setup, which means that the assembly cases have been constructed fairly quickly and without too much fiddling with parameters.

8.2. Use cases

To test the system, we use cases from industrial assembly manufacturing tasks, provided by the companies KVM-Conheat and VOLA. These assembly cases have not yet been automated by the companies. The cases are examples of small-batch productions and as a result, less time has been dedicated to the design of the hardware and mechanical setup. This implies that uncertainties are introduced based on imperfect gripper design, feeding and so forth. Consequently, uncertainties in the assembly process exceeds the assembly tolerances in both cases.

8.2.1. Case A: Pipe connection for heating system. The task is to assemble a pipe connection used for heating systems. The case is shown in Fig. 13(A). The connection consists of a brass tube and two union nuts with large tolerances. The assembly sequence for the case consists of the following steps: (1) Picking the tube. (2) Bending one end of the tube using an external machine. (3) Placing and repicking the tube from a device that spins the tube 180 degrees, thereby allowing the tube to be picked from the opposite end. (4) Picking and placing two union nuts on the tube. (5) Bending the other end of the tube. (6) Dropping the finished product in a box.

Physical setup: The pallet used for the case A is shown in Fig. 14. On this pallet is a set of magazine feeders from which the tubes and nuts are grasped. The pallet also contains a mock-up of a crimping machine used to bend the tubes. A pneumatic rotary gripper (SMC MRHQ10D-180S-N) is used to make fast and convenient 180 degree flips and re-grasp of tubes. Finally, a tool extension is made that fits in the hand of the Robotiq gripper. This is a pneumatic extension that allows the robot to grasp

https://doi.org/10.1017/S0263574717000613 Published online by Cambridge University Press

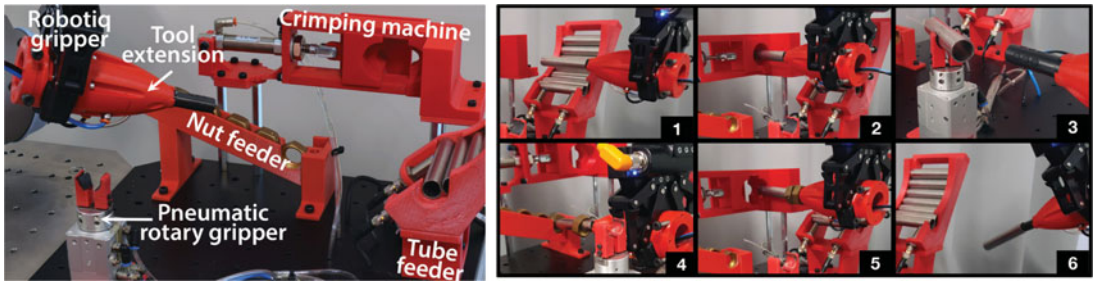


Fig. 14. Images series from selected parts of the case A assembly process.

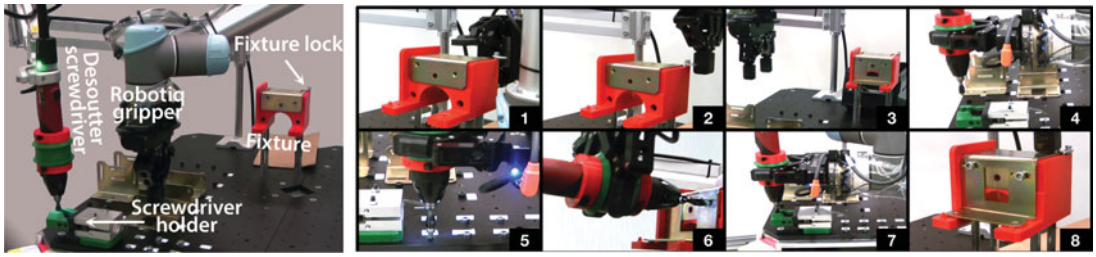


Fig. 15. Images series from selected parts of the case B assembly process.

the tubes from inside-out and to fixate the first union nut (such that it does not slide when grasping the second). Union nuts are picked from the feeder and attached to the tube in a single move.

Reversibility and programming: The reversibility of case A is limited in terms of the reversibility of the physical actions, due to restrictions posed by the feeders used and the pipe bending process. Of the six steps, one requires no modification, three require different indirect reverse behaviours, and the last two are not reversible. Even though the bending of the tube is a nonreversible process in real life, our mock-up machine does not perform an actual bending. The step was, therefore, used as a reversible step in our experiments.

Sensing and error detection: Based on a pre-analysis of the case A, it was found that in particular the peg-in-hole operation used for inserting the union nuts onto the tube was a challenging operation. The error detection system, therefore, mostly targeted this operation and the picking of the tube. Errors were detected by determining if the robots position matched the expected position of the pick operations.

8.2.2. Case B: Wall mount for wash basins. This case produces a subpart of a larger assembly of a wall mount for wash-basins. The case is illustrated in Fig. 13(B) and the assembly process is shown in Fig. 15. Assembly was performed in the following steps (matching the annotation in Fig. 15): (1) Pick and place a metal bracket. (2) Activate a lock on the fixture through a series of moves. (3) Pick and place a second bracket. (4) Pick the automatic screwdriver. (5) Pick a screw. (6) Place screw into the bracket (steps 5 and 6 are executed twice for two different screws). (7) Place the automatic screwdriver. (8) Unlock the fixture through a different series of moves, grasp the finished product and drop it into a box.

Physical setup: The pallet used for case B has a fixture in which the final assembled component is constructed. The fixture contains a manually operated lock that ensures the assembled part stays in place when detaching the screwdriver after having inserted a screw. This lock can be operated by the robot by pushing it into place. Brackets and screws are picked directly from the table. Brackets are placed on pre-specified places. Aligners help ensuring correct placement of the brackets that are picked directly with the robotic gripper. Screws are picked from a fixed position. Screws are magnetically attached to the screwdriver.

Reversibility and programming: The program splits case B into 16 operations. Nine of these require only the default inversion approach for reversibility, two require minor alterations and indirect reversible behaviour, and one was not reversible. The remaining four require a larger degree of modification, as they involved the more complex screwing operations illustrated in Fig. 6.

https://doi.org/10.1017/S0263574717000613 Published online by Cambridge University Press

Sensing and error detection: This case was difficult due to an unreliable and unpredictable screwing operation, where even small misalignments could cause the screwing operation to fail. In this case, errors were detected by monitoring I/Os from the screwdriver (which gives a signal if the screwing process deviates from a recorded process in either force or number of rotations). Errors were also detected by comparing the actual and expected position before and after the screwing process, which helped detect errors where the screw would be inserted skewed. The screwdriver was programmed through its own quick-setup program during which it learns from manual demonstration the number of rotations and the force to expect.

8.3. Experiment 1: Reversing the programs

To test the principle of reversible assembly, we use both use-cases for which we developed the corresponding programs. Forward execution performs assembly, while reverse execution should perform disassembly.

For each case, the program is executed forward to assemble an object. Afterwards, the finished objects were then manually placed back onto the gripper or fixture for cases A and B, respectively. The program was then executed backwards and disassembled the object. This was done a total of three times for each case.

Many assembly tasks are conceptually reversible (as discussed in Section 3), but reversibility is also influenced by the design and implementation of the assembly process and related hardware. Based on our cases, we discuss common findings related to this reversibility. This test also verifies that the error handling mechanism works for both the assembly and disassembly processes (running the program forward and in reverse). When an error occurred during the disassembly process, the program changed the direction of execution and resolved the error through forward execution before resuming disassembly.

8.3.1. Results. From these cases, causes that disrupt reversibility were identified along with operations that can only be reversed through indirect reverse processes. The main findings were:

Gravity: Many parts of the processes are significantly influenced by gravity, which often interferes with the reversibility of the assembly operations. Gravity is actively utilised to place and ensure correct positioning of objects. However, it disrupts the reversibility of operations by moving objects after release; hence, they will not be located where expected.

Feeding systems: Feeders are in general not directly reversible as they actively replace the picked object with a new. An object picked from a feeding system can, therefore, not be returned to the same position. An indirect reversible approach can, however, be created by discarding (or returning) the objects in a different manner before picking a new object.

Nonreversible operations: Case A contained a nonreversible operation in the bending of tubes. Even humans would require a significantly different set of tools for reversing this operation. Both cases contained a nonreversible operation in the final and last operation where finished objects were dropped into the box. It is observed that the frequency of nonreversible operations is fairly low in these two cases. Operations that are indirectly reversible due to limitations such as feeders, gravity, or other influences are more common.

Directly reversible operations: While both cases contained directly reversible operations, case B was found contain the most. This was mainly due to feeding systems method used. In our test, programs directly reversible processes made up 45% of all operations. Directly reversible operations such as the 'pick screwdriver' was used in both its forward and backwards form in the same program using the call and uncall functionality.

8.3.2. Discussion. We have found that both our use-cases could be made almost entirely reversible using either directly or indirectly reversible operations through the execution model and the programming language. If the reversibility concept was to be integrated more deeply into the design of assembly processes and external equipment such as feeders, an even greater degree of directly reversible instructions could be achieved.

8.4. Experiment 2: Assembling 100 objects

By assembling a large number of objects, we aim to demonstrate the use of reverse execution as an effective error correction tool. Results also provide an estimate of:

https://doi.org/10.1017/S0263574717000613 Published online by Cambridge University Press

- Which types of errors occur in small-batch assembly?
- Which error classes can/cannot be handled by reverse execution?
- How does the system as a whole, and the concept of reverse execution, perform?

8.4.1. Results. The workcell was set to assemble 100 objects of each type consecutively and without pause. During these 200 assemblies, a total of 22 errors occurred, of which 18, corresponding to 82%, were resolved and corrected using reverse execution.

During the assembly of the 100 pipe connections (case A), a total of 9 errors occurred of which 7 were corrected without manual intervention:

- Six errors were the result of the peg-in-hole operation used to insert the tube into the union nut jamming. Due to a collision, the tube failed to penetrate the nut.
  - Three of these were solved by backtracking a short distance before retrying the operation.
  - Three of these were solved by backtracking further before discarding the tube and repicking a new.
- One error was caused by the system dropping the tube during the turning operation (step 3). This was resolved by reversing until a new tube was picked.
- One error was caused by an air-tubing from the gripper getting stuck on the platform, causing the gripper to get misaligned. Recovery was unsuccessfully attempted, and required a manual reset and restart to correct
- One error was caused by the system failing to drop the finished product. This was a mechanical problem leading to an unrecoverable error.

During the assembly of the wall-mount (case B), a total of 13 errors occurred of which 11 were automatically corrected.

- Four of the errors were due to the system failing to grasp the screw. This was resolved by backtracking a short distance and repeating the grasp.
- Six errors were due to the screw failing to catch the threads of the brackets due to small misalignments. This was resolved by backtracking and repositioning the bracket.
- One error was due to the screw catching the threads incorrectly, causing the screw to be inserted askew. This was resolved by retrying the screwing.
- Two unresolved errors were caused by a screw being inserted at a skewed angle causing the bracket to misalign. In all cases, the errors were originally detected and the reverse execution was able to solve the problem with the misaligned screw and complete the assembly process. The system did, however, not correct the misaligned bracket, which is an undetectable problem in the current implementation.

8.4.2. Discussion. From the experiment, we see that reverse execution is capable of solving a wide variety of errors and that the exact method for solving each kind of error is not always the same. From our experiments, we see that the backtracking system is promising in handling errors related to small uncertainties in the assembly tasks, but that errors resulting in larger and mechanical failures still need to be addressed either in the design phase or by some other error handling mechanism. While the error classification shows the variety of errors that can occur, the actual numbers are only representative. The experiment is used to demonstrate rather than quantify the effects of using reverse execution as an error handler. The experiments also shows that while reverse execution can be used for solving a wide variety of errors, it also places strong demands on the error detection system.

9. Conclusion and Future Work

We have presented an analysis of reversible assembly and a generalized notion of reversibility. This is used to develop a concept of reversible assembly programming that allows programs to be executed both forward and backwards.

We show how reverse execution can be used for automatic error recovery, thereby increasing the robustness of the system. This demonstrates the first hypothesis in the paper, which is that reversible execution can be used to correct errors during robotic assembly programs. This will be, especially important in flexible assembly scenarios where an SME sets up its own workcell to produce

https://doi.org/10.1017/S0263574717000613 Published online by Cambridge University Press

low-volumes of products, where assembly errors will be more common than in professionally setup large volume scenarios.

The default mechanism for reversibility is through statement-level program inversion, which works well for directly reversible operations. However, an analysis of the assembly cases has shown that some operations require alternative reverse sequences of instructions. This is included in our model through the concept of indirectly reversibility. Finally, some operations are nonreversible, which is added to the model as well. These concepts have been used to demonstrate the second hypothesis regarding how a single program can be executed forward to perform assembly and backwards to perform disassembly.

To facilitate the creation of reversible assembly programs, we are developing the domain-specific language SCP-RASQ. SCP-RASQ allows overloading of otherwise nonreversible operations by extending the concept of reversibility into the programming language. In our language, the same operation sequence can be invoked both forwards and backwards. This is shown to enable additional code reuse and make programming of assembly tasks easier. To expedite the use of reversible assembly as an error handling mechanism, we have implemented reversible constructs for error detection and handling. The system as such is capable of handling and resolving a wide variety of errors.

Regarding reversibility, our system represents the first experimental demonstration of reversible computing principles applied to industrial robotics. Working with robots in real-world scenarios limits and challenges reversibility. In general, robotics is an interesting application area for reversible computing, with implications relevant for actual manufacturing scenarios, and implementing reversible languages for robotic control can contribute to a better general understanding of reversible phenomena, reversible computing and reversible programming languages.

In terms of future work, our short-term goal is to improve and ease the task of writing reversible programs for robot assembly. With a more advanced language and through better programming constructs, we aim to make it easier to specify differences in forward and reverse behaviour—even for complex situations. We also intend to provide features to improve trial-and-error capabilities. We plan to do this by making more operations fail gracefully and with reasonable errors. In the test-cases, the most prominent source of uncertainties and errors has been due to the tolerances and uncertainties in the montage process. To determine the exact uncertainties and how the method performs under other material, process, and montage tolerances remain an open question. An analysis similar to the one proposed by Buch *et al.*[13] could be used for this.

Acknowledgement

This work was supported by The Danish Council for Strategic Research through the CARMEN project. The authors acknowledge partial support from COST Action IC1405 on Reversible Computation. This work was initiated at Schloss Dagstuhl Seminar 15062 on Domain-Specific Languages.

References

1. Sparc, EU Robotics AISBL, *Robotics 2020. Multi-Annual Roadmap—For Robotics in Europe*, Brussels, (EU Robotics AISBL, 2014).
2. Robotics VO, *A Roadmap for U.S. Robotics from Internet to Robotics*, University of California San Diego, CA, United States (2016).
3. S. Jörg, J. Langwald, J. Stelter, G. Hirzinger, and C. Natale, "Flexible Robot-Assembly Using a Multi-Sensory Approach," *Proceedings of the 2000 IEEE International Conference on Robotics and Automation* (2000) pp. 3687–3694.
4. B. Zhang, J. Wang, G. Rossano, and C. Martinez, "Vision-Guided Robotic Assembly Using Uncalibrated Vision," *Proceedings of the 2011 IEEE International Conference on Mechatronics and Automation (ICMA)* (Aug 2011) pp. 1384–1389.
5. P. Neto, N. Mendes, R. Araújo, J. N. Pires, and A. P. Moreira, "High-level robot programming based on CAD: Dealing with unpredictable environments," *Ind. Robot: Int. J.*, **39**(3), 294–303 (2012).
6. H. Chen, G. Zhang, H. Zhang, and T. A. Fuhlbrigge, "Integrated robotic system for high precision assembly in a semi-structured environment," *Assem. Autom.* **27**(3), 247–252 (2007). doi:10.1108/01445150710763277
7. J. P. Buch, J. S. Laursen, L. C. Sørensen, L.-P. Ellekilde, D. Kraft, U. P. Schultz, and H. G. Petersen, "Applying simulation and a domain-specific language for an adaptive action library," *Lecture Notes Comput. Sci.* **8810**, 86–97 (2014). doi:10.1007/978-3-319-11900-7_8

https://doi.org/10.1017/S0263574717000613 Published online by Cambridge University Press

8. M. C. Koval, J. E. King, N. S. Pollard, and S. S. Srinivasa, "Robust Trajectory Selection for Rearrangement Planning as a Multi-Armed Bandit Problem," *Proceedings of the IEEE/RSJ Conference on Intelligent Robots and Systems* (2015) pp. 2678–2685. doi:10.1109/IROS.2015.7353743
9. T. Yokoyama, H. B. Axelsen, and R. Glück, "Principles of a Reversible Programming Language," *Proceedings of the Conference on Computing Frontiers (CF)*, New York, NY, USA: ACM (2008) pp. 43–54.
10. U. P. Schultz, J. S. Laursen, L.-P. Ellekilde, and H. B. Axelsen, "Towards a domain-specific language for reversible assembly sequence," *Proceedings of the 7th International Conference on Reversible Computation (RC 2015)*, Grenoble, France, July 16–17, (2015) (K. Jean and S. Jean-Bernard, eds.), Vol. 9138 (Springer International Publishing, 2015) pp. 111–126.
11. J. S. Laursen, U. P. Schultz, and L.-P. Ellekilde, "Automatic Error Recovery in Robot Assembly Operations Using Reverse Execution," *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2015).
12. U. Thomas, G. Hirzinger, B. Rumpe, C. Schulze, and A. Wortmann, "A New Skill Based Robot Programming Language Using uml/p Statecharts," *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)* (2013) pp. 461–466.
13. J. P. Buch, L. C. Sørensen, D. Kraft, and H. G. Petersen, "Facilitating Robotic Subtask Reuse by a New Representation of Parametrized Solutions," *Proceedings of the 13th International Conference on Informatics in Control, Automation and Robotics* (2016) pp. 37–48.
14. P. Loborg, Error Recovery Support in Manufacturing Control Systems *Ph.D. Thesis* (School of Engineering at Linköping University, 1994).
15. A. Christensen, R. O. O'Grady, M. B. Birattari, and M. D. Dorigo, "Fault detection in autonomous robots based on fault injection and learning," *Auton. Robots*, **24** (1), 49–67 (Jan. 2008).
16. D. Tarapore, P. Lima, J. Carneiro, and A. Christensen, "To err is robotic, to tolerate immunological: Fault detection in multirobot systems," *Bioinspiration and Biomemetics*, **10** (1), 016014–016014 (Feb. 2015).
17. R. Canham, A. Jackson, and A. Tyrrell, "Robot error detection using an artificial immune system," *Proceedings of the NASA/DoD Conference on Evolvable Hardware* (Jul. 2003) pp. 199–207.
18. S. Okumura, N. Take and N. Okino, "Error prevention in robotic assembly tasks by a machine vision and statistical pattern recognition method," *Int. J. Prod. Res.* **43** (7), 1397–1410 (2005).
19. Y. Wang, J. Cai, Y. Wang, Y. Hu, R. Xiong, Y. Liu, J. Zhang, and L. Qi, "Probabilistic Graph Based Spatial Assembly Relation Inference for Programming of Assembly Task by Demonstration," *Proceedings of the IEEE/RSJ Conference on Intelligent Robots and Systems* (2015) pp. 4402–4407.
20. C. Baydar and K. Saitou, "Off-line Error Prediction, Diagnosis and Recovery Using Virtual Assembly Systems," *Proceedings of the 2001 IEEE International Conference on Robotics and Automation (ICRA)*, Vol. 1 (2001) pp. 818–823.
21. J. Rosell, "Assembly and task planning using petri nets: A survey," *J. Eng. Manuf.* **218**, 987–994 (2004).
22. B. R. Donald, "Error detection and recovery for robot motion planning with uncertainty," *Technical Report* (Massachusetts Institute of Technology, 1987).
23. M. Fowler, *Domain-Specific Languages* (Addison Wesley, 2011).
24. Kuka Roboter GmbH, *Kuka System Software 5.5* (Kuka Roboter GmbH, 2010).
25. ABB Robotics, *Introduction to Rapid Programming – Operating Manual* (ABB Robotics, 2012).
26. A. Angerer, A. Hoffmann, A. Schierl, M. Vistein, and W. Reif, "Robotics API: Object-oriented software development for industrial robots," *J. Softw. Eng. Robot.* **4**, 1–22 (May 2013).
27. A. Nordmann, N. Hochgeschwender and S. Wrede, "A Survey on Domain-Specific Languages in Robotics," *Proceedings of the 4th International Conference Simulation, Modeling, and Programming for Autonomous Robots, SIMPAR 2014* (2014) pp. 195–206.
28. T. Buchmann, J. Baumgartl, D. Henrich, and B. Westfechtel, "Towards a Domain-Specific Language for Pick-and-Place Applications," *Proceedings of the 4th International Workshop on Domain-Specific Languages and Models for ROBotic Systems (DSLRob-13)* (2014).
29. J. S. Laursen, J. P. Buch, L. C. Sørensen, D. Kraft, H. G. Petersen, L.-P. Ellekilde, and U. P. Schultz, "Towards error handling in a DSL for robot assembly tasks," *Proceedings of the DSLRob '14* (2014) arXiv:1412.4538.
30. N. Hochgeschwender, S. Schneider, H. Voos, and G. K. Kraetzschmar, "Declarative Specification of Robot Perception Architectures," *Proceedings of the 4th International Conference on Simulation, Modeling, and Programming for Autonomous Robots, SIMPAR* (2014) pp. 291–302.
31. J. Huckaby and H. Christensen, "A taxonomic framework for task modeling and knowledge transfer in manufacturing robotics," *AAAI Workshops at the Twenty-Sixth AAAI Conference on Artificial Intelligence* Association for the Advancement of Artificial Intelligence, CA, US (2012).
32. A. Wahrburg, S. Zeiss, B. Matthias, J. Peters, and H. Ding, "Combined Pose-Wrench and State Machine Representation for Modeling Robotic Assembly Skills," *Proceedings of the IEEE/RSJ Conference on Intelligent Robots and Systems* (2015).
33. S. Bøgh, O. S. Nielsen, M. R. Pedersen, V. Krüger, and O. Madsen, "Does Your Robot Have Skills?" *Proceedings of the 43rd International Symposium on Robotics* (VDE Verlag GMBH, 2012).
34. R. E. Fikes and N. J. Nilsson, "Strips: A New Approach to the Application of Theorem Proving to Problem Solving," *Proceedings of the 2nd International Joint Conference on Artificial Intelligence (IJCAI' 71)* (San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1971) pp. 608–620. Available: `http://dl.acm.org/citation.cfm?id=1622876.1622939`

https://doi.org/10.1017/S0263574717000613 Published online by Cambridge University Press

35. C. R. Garrett *et al*. "Backward-forward search for manipulation planning," *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2015). Available: `http://lis.csail.mit.edu/pubs/garrett-iros15.pdf`
36. D. Hofmann, H. Huang and G. Reinhart, "Automated shape optimization of orienting devices for vibratory bowl feeders," **135**, 119–126 (Oct. 2013).
37. N. F. Edmondson and A. H. Redford, "Flexible parts feeding for flexible assembly," *Int. J. Prod. Res.*, **39**, pp. 2279–2294 (2001).
38. B. Stoddart, R. Lynas and F. Zeyda, "A virtual machine for supporting reversible probabilistic guarded command languages," *Proceedings of the International Conference on Reversible Computation (RC '09)*, Electronic Notes in Theoretical Computer Science, vol. 253, no. 6 (2010) pp. 33–56.
39. Zuliani, "Logical reversibility," *IBM J. Res. Dev.*, **45**(6), 807–818 (2001).
40. H. Agrawal, R. A. DeMillo, and E. H. Spafford, "An execution-backtracking approach to debugging," *IEEE Softw.*, **8**(3), 21–26, May (1991). doi:10.1109/52.88940
41. H. Mühe, A. Angerer, A. Hoffmann and W. Reif, "On reverse-engineering the Kuka robot language," *1st International Workshop on Domain-Specific Languages and Models for ROBotic Syst.* (2010).
42. FANUC, *Operators Manual – R-J3iC Controller Arc Tool* (FANUC, 2006).
43. ABB Robotics, "Rapid overview - technical reference manual," (2004).
44. U. Schultz, M. Bordignon, and K. Stoy, "Robust and reversible execution of self-reconfiguration sequences," *Robotica*, **29**, 35–57 (2011).
45. U. Schultz, "Towards a general purpose, reversible language for controlling self-reconfigurable robots," *Reversible Comput.* **7581**, 97–111 (2013).
46. S. Vongbunyong, S. Kara and M. Pagnucco, "A Framework for Using Cognitive Robotics in Disassembly Automation," *Proceedings of the 19th CIRP Conference on Life Cycle Engineering*, **19** (2012) pp. 173–178.
47. S. Vongbunyong and W. H. Chen, *Disassembly Automation*. (Switzerland: Springer International Publishing, 2015) pp. 25–54.
48. J. Szalatkiewicz, "Robotic Disassembly of the Waste of Electrical and Electronic Equipment, Based on the Criteria of Identification and Analysis of Waste Characteristics, Presented on the Example of Computer Hard Disk Drives," *Proceedings of International Conference on Product Life Cycle Management (PLM '11)* (2011) pp. 287–294.
49. H. Scharke, *Comprehensive Information Chain for Automated Disassembly of Obsolete Technical Appliances*, Schriftenreihe: Informationstechnische Systeme und Organisation von Produktion und Logistik. (GITO-Verlag, 2003).
50. L.-P. Ellekilde, J. P. Buch, D. Kraft, T. M. Iversen, J. S. Laursen, S. Mathiesen, L. C. Sørensen, H. G. Petersen, D. Chrysostomou, M. Hansson, and C. Schou, *Industrial Assembly Cases* (2016), University of Southern Denmark, The Maersk Mc-Kinney Moeller Institute, Denmark.

https://doi.org/10.1017/S0263574717000613 Published online by Cambridge University Press