Improving the Development Process for Teleo-Reactive Programming Through Advanced Composition

James Hawthorne, Richard J. Anthony, Miltos Petridis
School of Computing & Mathematical Sciences
The University of Greenwich
London, UK
{J.Hawthorne, R.J.Anthony, M.Petridis}@gre.ac.uk

***Abstract*—Teleo-Reactive programming as applied to autonomic systems is both a viable and exciting prospect as it inherently recovers from errors and unexpected events whilst proceeding towards its goal. It does this without the need to know in advance, the specific events which might occur, thus greatly reducing the human maintenance element. Whilst the benefits of this technique are great once the program has been composed, the challenge of validation attached to these programs also increases. We aim to ease the composition process needed when constructing T-R programs by building new abilities in to our existing Teleo-Reactive framework. Firstly, these new abilities will make it possible to detect validation issues at design time, thus reducing the likelihood of problems and increasing the ease of detecting them. Secondly, it will be possible for programs to be automatically composed from existing elements, thus further reducing the cost of validation issues. These ideas can even be extended to allow for dynamic composition and runtime changes to goals and actions.**

***Index Terms*—Teleo-Reactive Programming; Software Composition; Goal-Based Software**

I. Introduction

Teleo-Reactive (T-R) programs developed by Nilsson [1] were designed for autonomous control of mobile agents. T-R programs continually accept feedback from the environment, performing actions based on this current state. A T-R program is structured with a hierarchical list of condition and action pairs with each action fulfilling or partly fulfilling the condition of higher precedence. An action will end execution if it ceases to be associated with the highest true condition, either because the action has fulfilled the next condition or some other circumstance has caused this case. We have shown in previous work how T-R programs can be used to create reliable and robust autonomic solutions [2].

The Oxford English dictionary describes *Composition* as "the nature of something's ingredients or constituents; the way in which a whole or mixture is made up". This description could be imagined differently depending on the given allowable interaction of the context and the aim of the composition. For example, in a heavily component oriented software design [3], [4], most software engineers would imagine the links and interactions between the objects and components as being the *Composition* of the program. According to Szyperski [3], components "have to be carefully generalised to allow for reuse in a sufficient number of contexts" therefore the composition of the module could decide other labels it has attached. 'Reusable', 'modularised', or the antonym, 'procedural'.

When we talk about composition of T-R programs, the whole of the program is composed of its preceding rules leading to the satisfaction of the goal. The ordering of rules is fundamental in T-R programs as the higher the rule in the program, the higher the priority. In a procedural program, the modules of code are often tightly coupled to each other (if modules exist at all) and the composition of those modules are not greatly considered. The content of the modules take priority here. In component-heavy programs, the connections and interactions are considered to a greater extent, resulting in greater re-usability and portability. In this paper, when we talk about T-R composition, we are referring to the conditions each action is associated with and the order of the rules in the program list.

We have identified some of the problems in composing T-R programs [5]. It is easy to make a mistake in the logical design of the program such as incorrect rule ordering or an action needing an extra condition to proceed correctly. This could lead to perpetual skipping of one rule leading to falsehoods in another rule or that the program is deadlocked and can never reach its goal. In essence, T-R programs present a more natural and robust way of approaching a goal (natural in the sense that a human approaches an activity without first planning the millions of different events which could interrupt the process). This paradigm shift has however exposed new logic problems which are very easy to trigger and difficult to diagnose.

Very often in fact, a program may look entirely logical but when it is first run, mistakes become obvious to us. As an analogy, an incomplete diagram of a house may appear complete so an inexperienced builder may begin to build the house. However, a structural part of the house is not shown on the diagram and it is not obvious that it is needed either. After the house is built, the mistake is obvious.

In [5] we highlight some frequently occurring issues and propose a valuable set of guidelines to reduce the possibility of these issues from ever embedding themselves in the T-R program. We have already developed the Java Teleo-Reactive Autonomic Framework (JTRAF) to allow for the easy creation of high-level T-R solutions as well as allowing for future improvements and extensions such as the ones proposed in this paper to be applied without affecting existing JTRAF based programs. By building our solution into JTRAF, we can shift some of the validation work from the designer, tackle some of these logic problems and automate the T-R composition process. Thus T-R becomes a more viable autonomic solution.

Our proposal addresses two aims. The first, as already mentioned, is to detect validation issues with the program before deployment. This includes errors where an action causes two rules to become true at the same time. The lower priority rule is inadvertently made redundant because the highest priority rule is always executed before the lower one. Another error could be with deadlocked actions, where one action never causes a higher priority condition to change state and thus the logic can never proceed past this point. These are vitally important issues that are not easily detected or solved. From experience, we know that even simple T-R programs can contain problems which are not easy to diagnose through observation alone. Very often it is not until after an initial execution of the program, do we find that an action is deadlocked. This is more often than not, a fault in the high level logic than a code fault.

The second aim is a natural progression from the first. Namely, that if we can determine where faults are and if we can inform our program which actions contribute to the logical correctness of specific conditions, then it is intuitive that we should be able to automatically compose the program in terms of order and arrangement of rules and conditions. This will further reduce human maintenance costs.

II. Related Work

In [6] the authors describe a problem in T-R programs where actions are executed one at a time. They argue that actions should be eligible to be run concurrently with other actions as there may be periods where you do not want to cease one action to begin another. The work is proposed for the robotics domain for which T-R programs were designed. In one given example, “If a soccer robot should dribble and kick. If the robot stops dribbling in order to prepare kicking, the ball will roll away from the robot. Thus, the robot has to kick while it concurrently dribbles.” If a robot has a similar T-R program:

$$\begin{gathered} ..... \\ CanKick \longrightarrow Kick \\ HasBall \longrightarrow Dribble \\ ..... \end{gathered}$$

This may be a problem if the Kick action took more than a few microseconds to complete. In many scenarios it is possible to organise the logic so that actions are executed in a time-slicing / multi-process way, where two conditions are alternately switched between true and false states. The two actions would then give the illusion of concurrency.

Much of the work on T-R is aimed towards using a variety of artificial intelligence techniques to implement learning algorithms on the model. For example, [7] uses neural networks to capture new environmental experiences which can be integrated into a learning architecture. Whereas in [8] a representation formalism is developed called ‘teleoreactive logic problems’ which support learning. In this method, two knowledge bases exist containing a list of ‘percepts’ about the environment and a knowledge base containing known actions. They also describe an interpreter that utilizes the logic problems to achieve goals.

Through human guidance, the robots in [9] learn new T-R programs. These new programs are aimed mainly at navigation oriented tasks. To this end the low-level sensor readings are first transformed into higher level output so that they can be interpreted with physical objects represented as landmarks. Another learning algorithm is presented in [10]. The authors use genetic programming techniques to evolve T-R programs and produce programs for solving some problems.

We can think of a T-R system as a goal-oriented system as every T-R program proceeds towards its top level condition (the goal). There are several goal-oriented software designs which exist. Of particular note is [11], where a goal reasoning tool is built on the Tropos [12] methodology. Tropos itself supports software development at all stages, from requirements analysis to design with goal-based reasoning as a driving force. In [11], the authors try to make the goal analysis more complete by building forward and backward reasoning techniques into a graphical design tool.

This work is of particular interest to us because the backward reasoning employed is similar to the way in which our T-R software composition methods are implemented. The forward reasoning is employed to evaluate the impact of adopting the approaches with respect to *softgoals*. *Softgoals* in Tropos and indeed, in many other goal-oriented systems mean non-functional goals such as *Are customers happy?*.

In [13], the author argues the case for widespread use of Goal Oriented Requirements Engineering (GORE). He argues that it makes sense that the creation of software should be directed towards what the user wants to get from it, i.e. the goal, and that these goals should drive the requirements. The author shows evidence of several successful projects which use GORE and also argues that tool support should be integrated into GORE development.

[14] also champion GORE as a way forward in software development. They give a detailed account of the relationship between goals in GORE models and use several examples to illustrate their point. For example, the relationship between one goal and another goal or softgoal can have several satisfactory levels and contribution types, including satisfied, none, conflict, denied, some positive, some negative to name just a few.

It will certainly be interesting to see if our proposed techniques described in this paper can have their desired benefit with such minimal extra requirements on the T-R software designer considering the number of embellishments needed for the completion of a GORE-based system. However, the stages of development and their aims vary. i.e. GORE is largely

$$\begin{aligned}
IsFileComplete &\longrightarrow Nil\\
IsFivePacketsSent &\longrightarrow ChangePacketSize\\
IsConnected &\longrightarrow SendNextPacket\\
IsAccepting \wedge IsWaiting &\longrightarrow Connect\\
T &\longrightarrow Accept
\end{aligned}$$

Fig. 1. Typical view of a T-R program (simple file sender program)

concerned with modelling a system at the requirements stage whereas we want to aid the designer at the design/development stage.

Although T-R was developed as an autonomous low-level robotic and agent control method, we believe that the approach can be of use in higher level autonomic software and we are interested in applying this approach in this field. As such, we note some of this research.

There have been several approaches to autonomic software solutions, from architectural designs [15], [16] to artificial intelligence [17] and utility function based ideas [18], many of which are very complex solutions themselves and thus the ease of implementation is sometimes questionable. We wish to minimise "complexity tail chasing"; a term used in [19] to describe the situation where one aspect of the complexity problem is resolved by increasing complexity in other aspects. Such approaches can exacerbate the challenges of validation and trustworthiness.

III. Existing T-R Composition Method

For our discussion, we will use a very simple file sending T-R program first presented in [2]. This program was designed to show how the T-R system can recover from unexpected events. The program simulates a network file sending service from the point of view of the server. It sends a file in packets once a client has connected. Depending on the success of previous packet sending attempts, the packet size is adjusted accordingly; dynamically achieving a balance between communication efficiency and robustness.

A typical representation of this T-R program is shown in figure 1 with the boolean conditions to the left of the arrows, and the actions to the right. It is worth emphasising that the higher the rule, the higher the precedence. So lower rules are ignored if a higher condition is true. For example, in figure 1 if both *IsConnected* and a *IsFivePacketsSent* were both true then only action *ChangePacketSize* would be eligible to execute as *IsFivePacketsSent* is located higher in the program and therefore takes precedence (*IsFivePacketsSent* actually means has a *multiple of five* packets been sent 'true' or 'false').

This program was implemented using the JTRAF framework where each action contains only one condition. For example, the *Connect* action of figure 1 is instantiated in JTRAF with a reference to an instance of an *AndConditon* which itself contains references to instances of *IsAccepting* and *IsWaiting* conditions. Therefore connectAction only references one condition (*andCondition*).

```
                    .....
Conditional andCondition = new AndCondition(
     new IsAccepting(), new IsWaiting());
```

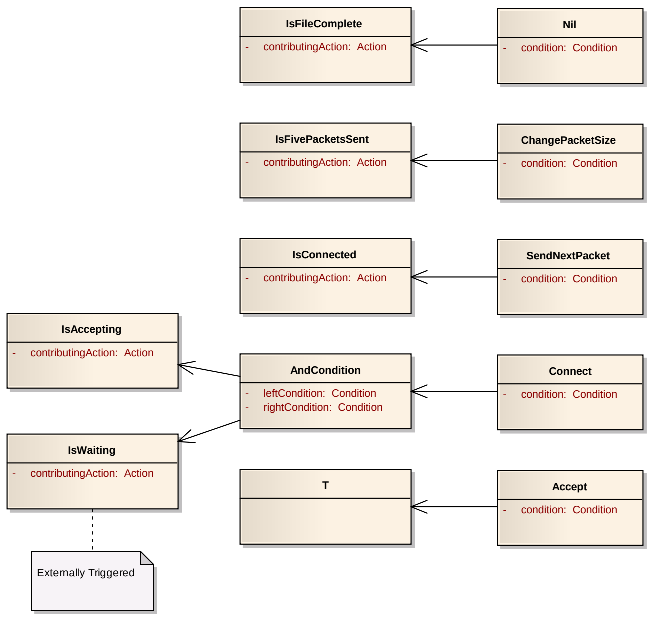

Fig. 2. Class diagram representation of the file sending program in JTRAF

```
Action connectAction = new ConnectAction(
     andCondition);
                    .....
```

Each action is instantiated in a similar manner but does not yet include the sequence order and the T-R program still needs to be composed. For the simple file sending application the code is as follows.

```
                    .....
TRProgram trProg = new TRProgram();
                    .....
trProg.addActionToBottom(nil);
trProg.addActionToBottom(changePacketSize);
trProg.addActionToBottom(sendPacket);
trProg.addActionToBottom(connectAction);
trProg.addActionToBottom(acceptAction);
                    .....
```

The method *addActionToBottom(Action)* is used by the program designer as shown to compose the actions in (what they perceive as) the correct order. In this case we create the T-R program of figure 1. A class diagram representation is shown in figure 2.

The important fact illustrated here is that, although each action has a reference to its conditions, it has no reference to any other part of the program. This could be considered good software design as the loose coupling means changes to one part of the program will have little effect on any other part.

A. Related Problems

The problem with T-R programs in general is the heavy reliance on the designer to correctly compose them. The order of rules and number of conditions related to each action and their arrangement have to be decided by the designer. This may appear straightforward but even in the simple file sending example mistakes can and were made. These mistakes not only invalidate the entire program but can be very difficult to find and diagnose.

As shown in figure 2 this heavy reliance is due to the fact that there is no link between the T-R elements within the program itself so the designer must arrange the elements in the order that is thought to be correct. What the designer thinks is correct and what is actually correct can be very different. Often, it is not until the program is first executed do these composition mistakes become evident.

As a real example, the program author (one of the authors of this paper) made the mistake of thinking that the packet size could be configured before packets were sent and originally reversed the order of the second and third rules. He neglected to recognise that organising the logic this way would make the packet optimisation rule redundant. As the packet sending rule now had priority over the packet optimising rule and since *sendPacket* only requires *IsConnected* to be true, the packet optimising rule will be skipped entirely. The goal of sending the file could still be achieved but without packet optimisation. Therefore it was difficult to ascertain that there was any fault in the program.

This example also serves to illustrate the difference in type of error which can occur in a T-R program and a program written in a high level language. We can imagine a similar program written in Java where any errors which do occur are likely to be a mixture of compile-time and run-time faults in code. Compile-time errors will be flagged by the compiler and corrected when they are first known. Run-time errors can be much harder to identify and are too numerous to have complete confidence that we have identified them all.

It is possible for an action to have an effect on more than one condition or a condition may or may not become true after only one iteration of its contributing action. If several iterations are needed; each successful iteration would be moving the program towards a state where the condition can become true. If a condition is made false by some action and is then prevented from ever becoming true again then the program is effectively deadlocked when *dropping* below this false condition as the associated action can now never execute and it may not be possible to reach the goal. Similarly, if a condition, once true, can never return to a false state then that may prevent a lower precedence condition from ever being reached. The associated action for the unreachable condition may be needed at a later stage by a different condition to make it true.

A T-R program can be correct despite not being able to predict its exact state at any moment due to its dynamic behavior which is sensitive to its operating context. This affords the T-R program the ability to recover from a broad range of unexpected events but the difficulty in building a formal model is exacerbated by the dynamic context-driven behaviour, as the sequence of states visited is not knowable pre-runtime and can be different in each run.

This highlights some of the complexity problems associated with composing a T-R program. Adding only one extra rule will likely result in an exponential growth in the complexity. The need for an automated method of detecting logical errors and composing T-R programs in order to expose T-R programming as a way to produce autonomic programs is proving to be a requirement rather than an option.

IV. Proposed T-R Composition Method

The T-R program designer will be required to make only one small addition to their program design. That is, to add a reference from conditions to actions informing JTRAF which actions contribute to the completeness of which conditions. By using this information to drive the composition algorithms we have implemented in the framework, JTRAF will be able to detect any logical errors in the T-R program design. With the newly supplied information and the current program composition it will be possible to determine if any condition is obtainable by recursively checking if the contributing action is located in a preceding rule. If so, we need to determine if this action will ever execute. We can do this by checking if the condition for this action is obtainable in the same manner as above. Essentially, we are forming a chain within the program as in figure 3 and thus be able to determine if the goal can be achieved or if any rule is skipped.

At the programming level we include a method (*isAchievable(Condition)*) which returns a boolean result indicating whether the provided condition will ever return true given the current composition of the T-R program. Since the goal of a T-R program is also a type of condition, the designer can test the goal and determine if the program is valid or not. If not, the designer can use the same method on the preceding conditions to determine the point of failure. For example in our previous file sender example, '*trProg.isAchievable(IsFileComplete);*' will return true because 'IsFileComplete' is achievable given the current program and additional contributing action information. However if the method returned false, the designer could work back through the program using this method in order to find the point of failure.

The *isAchievable(Condition)* method will warn the designer if for example the given condition 'is achievable' but some rules will be skipped entirely 'en route'. This indicates that there is an error in design but the condition is still able to become true.

As shown in figure 3 some conditions can be externally triggered. This means that the condition could be made true by some event external to the program itself. In the file sending case, *IsWaiting* becomes true when a client connects and is added to the waiting queue. Since we have no control within the T-R program over when this event might happen, we should mark the condition as being externally triggered. A condition should be marked as externally triggered if it is possible that the condition will become true at some point. A condition marked externally triggered when in fact it is always false could cause the program to be falsely validated as the goal might be perceived as reachable when in fact the incorrectly marked condition prevents this. In such a situation there is no guarantee that a condition will ever be triggered externally but this will not mean the program will ever fail

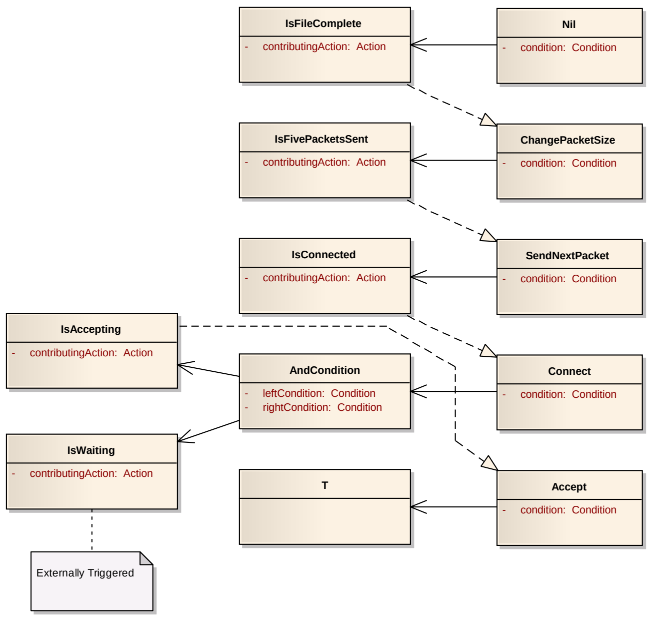

Fig. 3. Class type diagram representation of the file sending program in JTRAF with implemented composition design (solid lines indicate required conditions for an action, dashed lines indicate contributing actions for a condition)

entirely, only that the program will perpetually wait for the condition to become true.

The information required to detect composition mistakes is enough for the composition method to be extended, allowing JTRAF to automatically compose T-R programs as long as a composition is possible of course. Figure 3 shows the references between conditions and actions created through the contributing action variable and indicated by the additional arrows. The figure shows the program correctly composed and ordered. However, if the references exist, JTRAF will be able to reorder the rules and correctly compose the program even when the initial composition is incorrect or no previous composition exists. In any case JTRAF can return a correctly composed solution as in figure 3.

Again, at the programming level, a new method *compose()* will take the existing elements of the T-R program and arrange them so that the goal is achievable. Of course there may be more than one way to arrange the elements to satisfy the goal and the method may generate several failed compositions before a correct one is reached. The composition algorithm will implement the *isAchievable(Condition)* method as part of the mechanism for generating and testing for correct compositions. In this way the *isAchievable(Condition)* method becomes the foundation for composition, making *compose()* an intuitive and logical next step in validating a T-R program.

This short paper does not present the inner workings of the composition algorithm in detail, however, we provide a brief overview. For any condition to be achievable, it must hold a reference to a contributing action and the condition for this action must be satisfiable and associated with its own contributing action. This action must as well be associated with a satisfiable condition. This continues until the initial action and *T* condition are reached. If the solution is complete then it can be composed by JTRAF. This of course raises the possibility of multiple solutions, discussed in section VI.

Automatic composition has a big advantage for the program designer because it means that the logical ordering and numbers of conditions and actions and the problems of making a mistake here (see section III-A) are now less of a concern. If JTRAF cannot compose the supplied program, then it is likely to be because there is a lack of conditions or that the designer has neglected to add the contributing action to a condition. These problems are much simpler to rectify than the composition problems which occur without the aid of JTRAF's composition techniques. It also means that errors are discoverable at design time now. The designer does not have to wait until post deployment stage before composition problems are detected.

It is important to make clear that although information about which actions contribute to which conditions can now be provided, not every piece of information can be, and therefore the composition additions cannot guarantee the correctness of a program. Temporal data about condition checking and action execution might cause a deadlock but temporal data is not considered in JTRAF. This may cause a program to be validated whereas had the temporal data been considered and processed it would be impossible to do so. For example, suppose *ActionA* relies on *ConditionA* and *ConditionB* to be true. *ConditionA* is true when checked but it only remains true for two seconds. *ConditionB* takes three seconds before a true or false result is returned so by this time *ConditionA* is again false. If we knew this, then we could say that *ActionA* can never execute and therefore invalidates the program.

Entering temporal data into the composition algorithm would be of huge benefit in terms of validation for real-time applications but it would be unreasonable to expect precise timing measurements to be known in advance, especially for adaptive systems and/or systems with dynamic environments. A better solution would be for the measurements to be taken by and incorporated into JTRAF automatically. See section VI. It has always been an objective of JTRAF that any changes and improvements should not affect any existing programs which use JTRAF. Also, any additions which are warranted by the program designer should be easy to implement.

One of the main advantages of a pre-built Java framework is that the proposals in this paper and future methods can be seamlessly integrated into T-R designs. This means that the advantages are provided with the designer expending very little extra effort.

V. Conclusion

In this paper we have shown how effective and robust a T-R program can be and also how easy it is to compose the program incorrectly and thus never reach the goal. We have illustrated the seriousness of this problem and proposed that advanced composition techniques be built into our already existing Java T-R framework in order to automate the composition process, relieve the designer of this burden and to offer better assurance over the correctness of a program.

We believe that the techniques described in this paper are both necessary for the wide-spread use of T-R programming in both high and low-level applications. The solutions are implemented and benefits gained without requiring the program designer to endure much extra overheads. In fact to gain the benefits of advanced, automated composition, the designer is required to complete only one extra variable in order to link conditions to contributing actions.

VI. Future Work

There is a lot of scope here for future ideas to be implemented. In essence the work presented in this paper forms the foundation for many adaptations.

For example, if a condition has more than one action which can contribute to it; JTRAF could use this information to form multiple solutions for the completion of the program goal with the different solutions using the different possible actions. It may be that some of the actions contribute to the condition in question but not to another higher priority condition, which may be necessary for completion of the goal and therefore this particular solution is invalid. This information about the exact correctness of the composition might only be discoverable at run-time, since the program might need to be in operation before some information is known. For instance, the designer might claim that *ActionA* contributes to *ConditionA*, but after the program has been in operation, it is automatically discovered that this is not the case. JTRAF could then *self-adapt* according to an actual correct composition. The developers' original composition of the program could be treated as a suggestion rather than an unchangeable structure, with the composition always assessed and changed by JTRAF at run-time according to the proven reliability of each composition.

It may also be true that several of the actions are applicable to a particular program i.e. several actions could be used in the program to complete the same goal. In which case, JTRAF should determine which of the solutions is the better one. Perhaps one solution completes the goal in the quickest time whilst another solution is more reliable i.e. the actions in the reliable solution are much slower but rarely contain errors.

The failure of an action is less of a concern in a T-R program since if an action fails, all this means is that the program will fall back to a lower precedence rule until it is ready to try the action again. Never-the-less the program designer may require some actions to be more reliable than others. Perhaps using a combination of weights in a utility function and policies for the designer to decide whether speed or robustness is more of a concern to them.

We could also adapt JTRAF so that actions and conditions can be dynamically inserted into a running T-R program. Either a human could insert new conditions and actions or JTRAF could learn a new action by itself and dynamically insert it into the program. These newly learnt actions could even be used in the composition of completely new goals which were not possible before the new action existed. With the advanced composition techniques implemented there would be no need to decide where to place the new condition or action. JTRAF could decide where in the program it should be placed.

References

[1] N. J. Nilsson, "Teleo-reactive programs for agent control," *Journal of Artificial Intelligence Research*, vol. 1, pp. 139–158, 1994.

[2] J. Hawthorne and R. Anthony, "Using a teleo-reactive programming style to develop self-healing applications," in *Autonomics 2009: Third International ICST Conference on Autonomic Computing and Communication Systems*. Limassol, Cyprus: ICST, September 2009.

[3] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[4] C. Szyperski, "Component technology: what, where, and how?" in *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 684–693.

[5] J. Hawthorne and R. Anthony, "A methodology for the use of the teleo-reactive programming technique in autonomic computing," in *Software Engineering Artificial Intelligence Networking and Parallel/Distributed Computing (SNPD), 2010 11th ACIS International Conference on*, June 2010, pp. 245 –250.

[6] G. Gubisch, G. Steinbauer, M. Weiglhofer, and F. Wotawa, "A teleo-reactive architecture for fast, reactive and robust control of mobile robots," in *New Frontiers in Applied Artificial Intelligence*, ser. Lecture Notes in Computer Science, N. Nguyen, L. Borzemski, A. Grzech, and M. Ali, Eds. Springer Berlin / Heidelberg, 2008, vol. 5027, pp. 541–550.

[7] J. Ramírez, "Neural synthesis of teleo-reactive programs," in *ICMAS '98: Proceedings of the 3rd International Conference on Multi Agent Systems*. Washington, DC, USA: IEEE Computer Society, 1998, p. 459.

[8] D. Choi and P. Langley, "Learning teleoreactive logic programs from problem solving," in *Proceedings of the Fifteenth International Conference on Inductive Logic Programming*. Springer, 2005, pp. 51–68.

[9] B. Vargas and E. Morales, "Learning navigation teleo-reactive programs using behavioural cloning," in *Mechatronics, 2009. ICM 2009. IEEE International Conference on*, 14-17 2009, pp. 1 –6.

[10] M. J. Kochenderfer, "Evolving hierarchical and recursive teleo-reactive programs through genetic programming," in *Programming, EuroGP 2003, LNCS 2610*. Springer-Verlag, 2003, pp. 83–92.

[11] P. Giorgini, J. Mylopoulos, and R. Sebastiani, "Goal-oriented requirements analysis and reasoning in the tropos methodology," *Engineering Applications of Artificial Intelligence*, vol. 18, no. 2, pp. 159–171, March 2005.

[12] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos, "Tropos: An agent-oriented software development methodology," *Autonomous Agents and Multi-Agent Systems*, vol. 8, no. 3, pp. 203–236, 2004.

[13] A. van Lamsweerde, "Goal-oriented requirements enginering: A roundtrip from research to practice," *Requirements Engineering, IEEE International Conference on*, vol. 0, pp. 4–7, 2004.

[14] D. Amyot, S. Ghanavati, J. Horkoff, G. Mussbacher, L. Peyton, and E. Yu, "Evaluating goal models within the goal-oriented requirement language," *Int. J. Intell. Syst.*, vol. 25, pp. 841–877, August 2010.

[15] E. M. Dashofy, A. van der Hoek, and R. N. Taylor, "Towards architecture-based self-healing systems," in *WOSS '02: Proceedings of the first workshop on Self-healing systems*. New York, NY, USA: ACM, 2002, pp. 21–26.

[16] J. Kramer and J. Magee, "Self-managed systems: an architectural challenge," in *FOSE '07: 2007 Future of Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 259–268.

[17] S. Hassan, D. Mcsherry, and D. Bustard, "Autonomic self healing and recovery informed by environment knowledge," *Artif. Intell. Rev.*, vol. 26, no. 1-2, pp. 89–101, 2006.

[18] J. O. Kephart and R. Das, "Achieving self-management via utility functions," *IEEE Internet Computing*, vol. 11, no. 1, pp. 40–48, 2007.

[19] R. J. Anthony, "Policy-centric integration and dynamic composition of autonomic computing techniques," in *ICAC '07: Proceedings of the Fourth International Conference on Autonomic Computing*. Washington, DC, USA: IEEE Computer Society, 2007, p. 2.