Using a Teleo-Reactive Programming Style to Develop Self-healing Applications

James Hawthorne and Richard Anthony

Dept. Computer Science,
The University of Greenwich, London, UK
{J.Hawthorne,R.J.Anthony}@gre.ac.uk

Abstract. A well designed traditional software system is capable of recognising and either avoiding or recovering from a number of expected events. However, during the design phase it is not possible to envision and thus equip the software to handle all events or perturbations that can occur; this limits the extent of adaptability that can be achieved. Alternatively a goal-oriented system has the potential to steer around generic classes of problems without the need to specifically identify these.

This paper presents a teleo-reactive approach for the development of robust adaptive and autonomic software where the focus is on high level goals rather than the low level actions and behaviour of software systems. With this approach we maintain focus on the business objectives of the system rather than the underlying mechanisms.

An extensible software framework is presented, with an example application which shows how unexpected events can be dealt with in a natural way.

Keywords: Robust software, Goal-based systems, Software frameworks, Error recovery, Context awareness, Self-healing.

1 Introduction

The process involved in achieving a goal for a human or other living creature is very different to the way in which a goal is achieved in a computer programming language. The level of robustness in a computer program is determined by software developers, in the sense that errors can be prevented or caught by a system of try-catch exception blocks or if-else statements. However, the programmer must explicitly implement these techniques and it is very easy to miss some errors or catch and deal with one error only to fail to deal with that same error if it reoccurs whilst dealing with it at a different level in the code; also there is the decision as to which error to deal with at the point they are detected, and which need to be thrown up to some higher level handler.

In short, there are many ways for many types of errors to cause failure and it is almost impossible for a programmer to deal with them all (some of which were not even known at design time) using these built-in techniques. The problem is that the types of standard techniques and methods available to programmers offer a quite un-natural way of producing robust systems.

A.V. Vasilakos et al. (Eds.): AUTONOMICS 2009, LNICST 23, pp. 114–129, 2010.
© Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering 2010

Consider how humans achieve a goal; for example, loading the dish washer. We have several items to load into the dish washer and we know that if the dish washer door is closed we must first open it. So opening the dish washer could be considered a sub-goal towards the main goal of loading the dish washer.

Next, we consider if there are items already in the machine so we must take the clean ones out first. We must then put the heavier items in the bottom sturdier drawer and the lighter items in the top drawer but your partner unintentionally puts a large plate in the top drawer which means you now have to take out that plate and put it in the bottom drawer before continuing to load the rest in. You are then about to place a cup in the dish washer but your partner does this task before you get a chance to pick up the cup. This means that you are no longer concerned with that cup, so continue with the other items. Humans continue to monitor the situation like this so that we can prioritize tasks in this way. We continue until the goal of loading the dish washer is complete and all the items are loaded.

As another example, you can hear someone calling your name behind you, but you do not exactly know their position. If you continue to look round you will eventually see the person and can respond. You know the general direction of this person but there is no discrete instruction, like rotate 167 degrees as there might be in a computer program. We just continue to rotate until the goal "Can see person" is met.

Within this human process there are many low level decisions made and many possible variations on the actual sequence of turning and looking, yet this seemingly straightforward task (with all its inbuilt contextual adaptation) is very difficult to express using traditional procedural or object oriented code - even if well structured.

Teleo-reactive (T-R) programming [1] was initially developed for the robotics domain but the techniques involved produce systems which react and resolve problems in a more natural way, similar to how humans head towards a goal (with continuous context aware micro-adaptation). With T-R programs, a system heads towards a goal without knowing how to get there precisely. The basic methods involved in these programs produce more robust code which can gracefully recover from errors or unexpected events.

From the users point of view this gives the impression of self-healing. Though faulty code is not strictly replaced with 'fixed' code, the program will continue to run after an unexpected event or error occurs. T-R programs effectively 'fall back' to an earlier stage and work back to this point. The term 'Self-healing' is used in this work to mean a recovery and continuation from an unexpected event, rather than a physical fix or replacement of faulty logic. A more in-depth description of how this works is provided in section 3.

T-R programs have been shown to be highly effective in building robotic agents, but in higher level systems it has not been extensively tested. A Java based software framework variation of a T-R system and example implementation have been built to allow exploration of the feasibility and benefit of using T-R programs in higher level systems. The example implementation described in

this paper shows how we can write software without the need to know precisely how individual goals will complete and that we do not need to know exactly what errors will occur to deal with them.

In fact, we find that to some extent, low level error handling is replaced by higher level goal tracking logic. This is important because goal tracking seems to be a lot more intuitive for humans to express accurately than exhaustive error identification and handling. Thus we propose this should be the foremost aspect during software development.

The rest of the paper is organized as follows. Section 2 discusses similar work, including work which achieves similar results to our own with different methods and techniques. We also highlight some of the varied domains in which T-R programs have been applied. Section 3 gives an overview of the components of our framework and how to apply them. We also demonstrate the difficulty of achieving similar results with traditional approaches. Section 4 shows the levels of design involved in the construction of programs using our framework. We show some of the design features and the simplicity of our framework. We then extend the framework to produce a simple example in section 5, show some of the ways we can extend the framework in section 6 and conclude in section 7.

2 Related Work

In our framework and in T-R programs in general, the focus is on *goals* as the most influential part of a program. Maintaining focus on these high-level aims is essential in delivering a valid product. Goal-oriented requirements analysis and reasoning is the main subject of [2] who use the Tropos methodology [3,4] to make goal analysis more complete, developing a formal model for this aim. The authors have developed a goal reasoning tool, which allows algorithms for forward and backward reasoning to be run on goal models. The backward reasoning in Tropos [3,4] is used to analyse the goal models to find the minimum cost goal that could guarantee the achievement of top level goals. This could be important for our framework as we try to guarantee goal achievements.

[5] places a high degree of importance on high level goals with the focus on functional and non-functional goals, such as performance and quality of service. We have mainly been targeting functional goals although our framework could be used to address non-functional issues as well. The authors of [5] also focus on how to generate these goals in the first place, saying that many goals can be obtained simply by asking HOW and WHY questions to obtain parent and sub-goals. Simple Use Case diagrams described in [6] are a good way to obtain and focus on initial goals.

The current work on self-healing systems has largely been directed towards enabling learning and dynamic updates to find better fitting solutions to problems. This is a logical method because the main goal of autonomics is to reduce the reliance on human assistance. However, many of these models and architectures offer very heavyweight solutions to self-healing or only offer a finite selection of alternative models.

As an example [7] uses a large number of connected components to support adaptation. The required changes are simulated to verify the proposed changes. If successful, the changes are merged using an architectural 'diff' tool. [8] also uses a complex model of interconnected components to monitor, analyse, perform adaptation etc. The model is 'externalized' from the running system providing a way to monitor and understand the system in a high level way. This is quite similar to many meta-modelling approaches where the level of abstraction allows changes to be made without the need to know the precise details of the program code.

In contrast to these self-healing systems, [9] uses case and rule based reasoning to make a correct decision about how to proceed in the case of problem events. Storing and retrieving a solution from the case-based-reasoning (CBR) system if it has previously been encountered, or contacting the domain expert if not. They use the example of diagnosing a faulty printer. However, many printer problems require human intervention such as adding more paper to the tray or replacing an ink cartridge and in many situations the system asks the user some questions to help it find the fault. In several cases a technician is called to make the fix. Although this technique could be useful for these 'non-automated' problems, self-healing is usually applied to software oriented errors and the recovery from them.

2.1 Teleo-Reactive Programs

T-R programs developed by Nilson [1] are 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 immediately higher precedence. An action will end execution if it ceases to be the highest true condition, either because the action has fulfilled the next condition or some other circumstance has caused this case. Figure 1 is a graphical representation of this production rule structure as shown in Nilsson's work.

This structure directs a design so that the top level condition (the goal condition) of a program is worked towards and will eventually be satisfied. The robot

$$
\begin{array}{ccc}
K_1 & \rightarrow & a_1 \\
K_2 & \rightarrow & a_2 \\
 & \cdots & \\
K_i & \rightarrow & a_i \\
 & \cdots & \\
K_m & \rightarrow & a_m
\end{array}
$$

Fig. 1. Condition-Action production rule list. K_i are conditions and a_i are actions. Condition-Action rules are evaluated from the top down, with the higher rule taking precedence over a lower one.

example shown in [1] and block stacking example applet shown at [10] further illustrates this technique.

The author of [11] has produced a reasoning framework, supporting verification of T-R programs. This is especially important in the case of safety and mission-critical software, but it does increase the complexity in developing T-R programs, unnecessarily in many cases as the system is quite robust in the first instance.

Basing their approach on T-R programs, [12] have designed GRUE, an architecture for controlling game characters (agents). With GRUE the agents are able to react to competing goals with conflicting requirements. GRUE is able to generate new goals in response to the changing current situation and allows multiple actions to be run in parallel in pursuit of several simultaneous goals, thus producing more convincing and believable game agents. The work supports the idea that the T-R approach not only allows programs to be written more naturally, but also produces more natural behaviour from agents.

3 Method

The example application presented in section 5 was created from a base framework. This framework is a generic architecture designed to make it simple to design applications using the methods of T-R programs and gain self-healing benefits. The example extends the actions and conditions in this base framework as required for the specific program needs. Each action must evaluate its condition and obtain a positive result before it is permitted to execute. The actions of the program should build on one another. That is, an action, once executed, should complete or go some way to completing part of the condition of the next action. This action may be executed more than once before the next condition is satisfied. Once this next condition is evaluated to ‘true’ its associated action can be executed. This action builds towards completing the next condition, and so on until the program is complete.

One important difference between our framework and regular T-R programs is that a condition which fails for whatever reason will return ‘false’ when queried. Designing code this way, produces self-healing behaviour; for example, a condition which fails for whatever reason will produce a negative evaluation and the program will be forced to ‘drop’ to a lower level. If the lower condition fails, the program will drop back further, until a true condition can be evaluated. Evaluating these conditions and executing the corresponding actions in order will ‘build’ the program back up to the original failing condition. On this next attempt, the condition should be ‘true’. If not, it will continue with the previous action until it is. In the same way, if a higher priority condition becomes ‘true’, then there is no longer a need to perform the previous steps and the program will continue from the highest priority ‘true’ condition-action pair.

This method is a more natural way to produce code that is aligned closely with the way humans solve problems. We continually monitor the ‘state of play’ to determine what our next move should be. We will not simply fail if we meet a

problem. If we need to complete a goal, we often perform a simpler action first, which makes the more complex action easier to address when we meet it again. The TR approach is similar but it is important to understand how this method works in order to effectively use the framework.

Following are short descriptions of the framework elements and classes.

3.1 Goals

In the framework there is no specific type (class) 'goal', but the highest priority condition is effectively the goal, i.e. all the lower prioritized actions work directly or indirectly towards it which marks the end of the program. Another way to view goals is that each T-R program is itself a goal and each T-R program consists of actions and conditions as shown in section 4.2.

We should make clear that the goals should be described in a high level manner, for example a condition 'Has connected client' in the server side of a network application can be treated as a simple boolean condition, but the underlying code to perform this query may be complex and technical. The check might involve sending small packets back and forth to verify that a connected client is still 'alive' or we might just be checking a list of clients which were previously connected and presumed to be still connected. In the later case, more errors are possible later as there is no 'pre-emptive' checking as in the first case. In which case a well designed T-R program will recover. The T-R approach supports functional division of logic and modular development, for example there could be two levels of programmers for the application, non-technical ones to write the T-R program in terms of high level goals and technical ones to provide the low level implementation.

3.2 Conditions

Conditions are linked tightly to actions. An action will only be executed when the associated condition is true. The action directly preceding should work to 'complete' this condition. As an example, a condition could check that a link to some external database is established before its action to gather some records is executed. The previous action would work towards establishing that link and therefore the action to gather records will not be executed until the connection is established.

A condition will return false if an error occurs whilst it is processing. This prevents the program from failing when an error is encountered. In this case the T-R program 'drops' to a lower condition and it is likely that the previous action will be re-performed. If we knew exactly what the error was then we could deal with it directly but to catch and deal with all errors we must perform some broader scoped operations. In short, we do not need to know what the error is to be able to recover from it, but the recovery process might need to go back more levels than is strictly necessary. An analogy would be to replace your car engine because there is an unusual noise coming from it. It would fix the unknown problem that we might not be able to fix ourselves, but it is a big change just

to fix a possibly small problem (fan belt just need tightening perhaps). A T-R program of finer granularity would likely be able to perform simpler actions to recover from an unexpected event.

3.3 Actions

Actions in a T-R program are performed only when the corresponding condition is true. Actions are arranged within a T-R program in order of ‘closeness’ to a goal condition. This way a T-R program builds towards this goal.

Actions will gracefully exit if an error occurs. In such cases, the program will be evaluated again from the top condition down. It may be the case that the error is a rare event and the action will likely succeed when met again. It may also be the case that the action took a relatively long time to complete and the condition which was true before execution is no longer true (a resource is no longer available perhaps). In this case the previous action will likely restore the resource and the action can ‘try again’.

It is important to state that actions could and should have simple names but the function of which can be quite involved and complex and could even be constructed with other TR programs. For example, an action called ‘Draw Circle’ in a graphics application might make several low and high-level calls to a graphics object and driver to perform the action and we should take this approach to naming and constructing actions when writing a TR program.

3.4 Contrast

Here we consider methods (available in most programming languages) for dealing with and avoiding errors. The simplest alternative method to avoiding errors would be a simple conditional statement. Consider a situation where we want to write some output to a stream, but we realize that if the stream has not yet been initialized it could cause problems:

```
if (printStream != null)
   printStream.write("Hello World!!");
```

The if statement avoids using the print stream if it has not yet been initialized. This is fine if that were the only possible error in the system, but this is very unlikely to be the case. As we recognize some other possible errors, we add these to our checks and end up with a large ball of spaghetti code, an untraceable structure consisting of nested if-else statements. Alternatively we could use an exception handler to catch any error within the try block. Something like this:

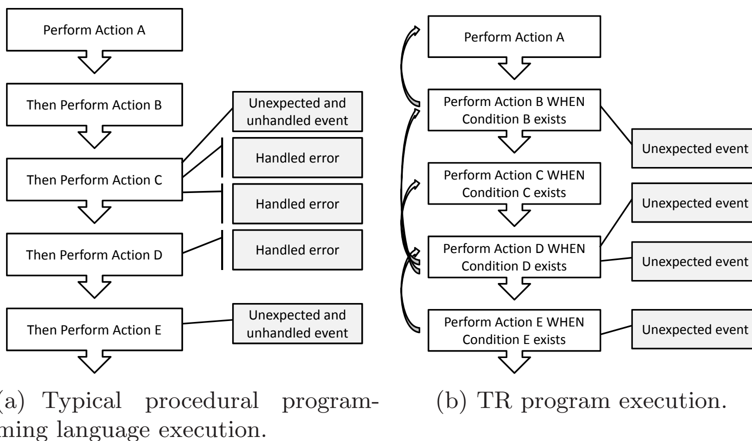
```
try{
   printStream.write("Hello World!!");
}
catch(Exception e){ \\ignore error }
```

The problem of not initializing the stream before we use it and any other problems would be handled and in this case they would be ignored. This might

catch problems, but it does not implicitly fix or recover from the problem. We can add extra code within the catch block to reinitialize the print stream, but again the problem is only solved for one instance of one type of problem and there is no guarantee that this, now fixed code, will ever be accessed again. The next steps after the handler might cause the stream to become uninitialized, but because this code is no longer within the try-catch block, the error is no longer caught. It would be difficult and time consuming to write exception handlers at every point in the code where an error is possible (and clearly the fact that so much commercial software is released with 'bugs' is evidence that even very large software houses are unable to develop code to explicitly handle all possible faults).

Another problem with traditional approaches is that it is easy to assume that the problem, handled once is no longer a concern. This requires the handler to be either duplicated or called from all places the problem could occur (one of the authors previously worked at a software house who released products typically containing hundreds of 'known' bugs - on the basis that most of these had very minor impact and were 'unlikely' to occur, being triggered during very specific sequences of events, and were thus very costly to track down and fully eradicate). In contrast, the T-R approach deals with an error whenever and wherever it occurs at any level of seriousness and for as many times as it arises.

Figure 2(a) highlights the problem in procedural languages of dealing with an event which the programmer had never thought of as a possibility (if the event is not envisioned how can it be dealt with?). What usually happens is the program 'crashes' and the programmer(s) spend some time finding the cause of the problem. The 'bug' is usually fixed but the fix may have caused another error (perhaps more serious than the first) at some other point in the code and it is likely that there are still many more errors to be revealed.



(a) Typical procedural programming language execution.

(b) TR program execution.

Fig. 2. Comparison of typical event and error handling processes in procedural and TR code execution

In contrast figure 2(b) shows that a TR style program executes each stage only when its condition is met. The program will not execute a stage if its condition is not fulfilled and will instead return to a previous step, perhaps executing previously performed code until the condition has been satisfied. The same steps are performed in the case of an unexpected event. Possible unexpected events are indicated in the diagram by the labels on the right and possible roll-backs are indicated by the arrows to the left of figure 2(b).

4 Design

There are effectively two stages of design when using the T-R method and framework. The first is the T-R program part and the second is the more familiar program structure design.

4.1 Teleo-Reactive Program Design

This part of design is the most important part, because it describes the flow of execution of the program. The design of the structure should be more obvious once this part is complete, although it is likely that this part of design will be modified through iterations of the system design phase.

There are some guidelines to follow when designing this part of the program which should make this phase a bit easier. Essentially we need to decide what actions and conditions there should be and in what order. The first condition should be the TRUE condition and the last condition should represent the goal.

The TRUE condition will always return 'true' and thus its action will always execute if there are no higher priority 'true' conditions. This means that if no other action is possible, the action associated will execute until one of the higher priority actions are possible. The TRUE condition provides a place where execution can begin, a definite starting position.

The final goal condition should be the state which all other actions are working towards. Here is an example where a delivery company uses GPS to track its delivery van and display the status of the van on a web site for customers to view (the software is running on a computer in the van depot):

$$\begin{aligned}
Has_van_returned_to_depot &\longrightarrow Nil \\
Has_DB_link \wedge Has_van_status &\longrightarrow Write_van_status_to_DB \\
Has_van_status &\longrightarrow Get_DB_link \\
Has_van_link \wedge Timer_expired &\longrightarrow Get_van_status \\
T &\longrightarrow Establish_van_link
\end{aligned}$$

Now imagine we are at the stage where we about to write the van status to the database, but the 'Has van status' condition is no longer true for whatever reason. The program may 'drop back' to establishing the link to the van and getting its state again. From this point it can work back to the point of writing to the database, thus the program self-heals.

The framework allows each T-R program to be run as a thread so in this case the T-R program would have as many instances as there are vans to track, each running in the background, leaving the main program free for more important tasks. We can be confident that each instance will work unsupervised because we know errors will be handled. This is why confidence in this part of design is so important. This is important for autonomics applications, where programs need to handle themselves resulting in little maintenance for the programmer after deployment, and only low levels of external supervisory input are needed.

4.2 Program Structure Design

The framework has been designed to allow the concepts discussed in this paper to be realized and even integrated into current systems as easily as possible. Programmers are required only to extend the types *Action* and *Conditional* as shown in figure 3. It is important to recognize that the onus currently resides with the programmer to produce actions and conditions which can complete at least some of the time, given the expected state and context. The framework robustly handles events and errors, however, an action or condition which never completes will always cause a 'deadlock' situation and the T-R program will never progress past this point. In section 6 we discuss how future iterations of the framework might reduce or remove the chances of this problem occurring.

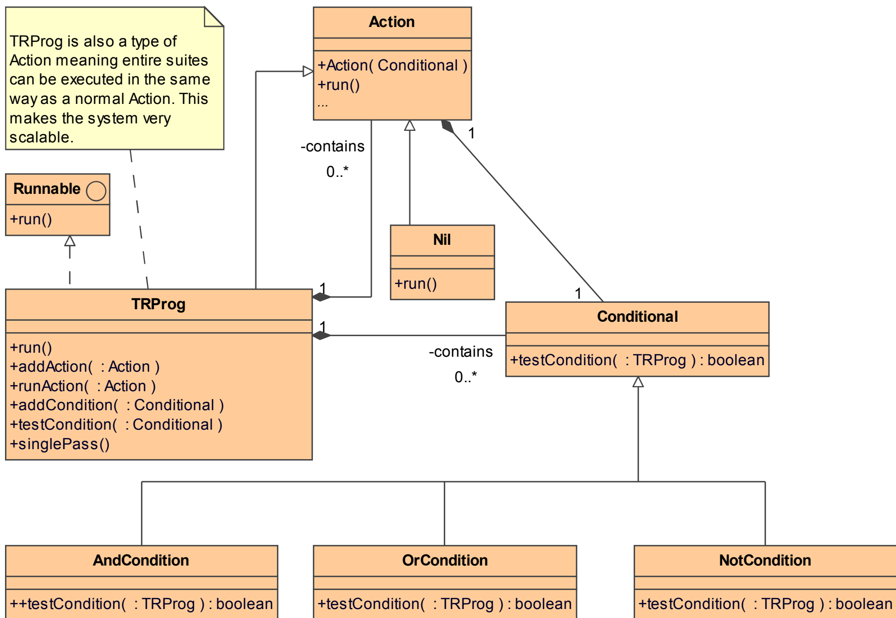


Fig. 3. Main framework presented as a UML class diagram

The design allows for easy modifications and extensions to be made to the framework without affecting current implementations. For example future versions of the framework could prohibit an Action which causes a deadlock from ever being added to the T-R program (See section 6).

The TRProg type is used as the main controlling class of the framework. This type contains an ordered list of Actions and an unordered list of Conditionals. Actions are executed in the order they appear in the list if its associated Conditional type returns TRUE. If not, the TRProg will 'try' the next Action. Conditionals are automatically added when the Action is added to TRProg unless the Conditional instance is already listed.

As can be seen in figure 3, TRProg extends the Runnable interface, meaning a TRProg can be run as a background thread. This can be useful to enable a T-R program to asynchronously run in the background performing some vitally robust task while your main thread of execution continues unaffected.

The figure also shows that the TRProg is itself a type of Action, meaning that an Action in the list could be an entire TRProg type, containing a different set of Actions. The TRProg may also call itself in a recursive fashion.

Actions that have been extended are required to implement the Run() method. If its condition evaluates to true, the code in this method is executed. The provided Nil Action is an empty action and is usually associated with the top level goal condition, i.e. if the goal condition is reached, perform no action.

When an action is constructed it must accept exactly one Conditional argument. However, this single conditional can contain many sub-conditionals when layers of And, Or and Not types are nested.

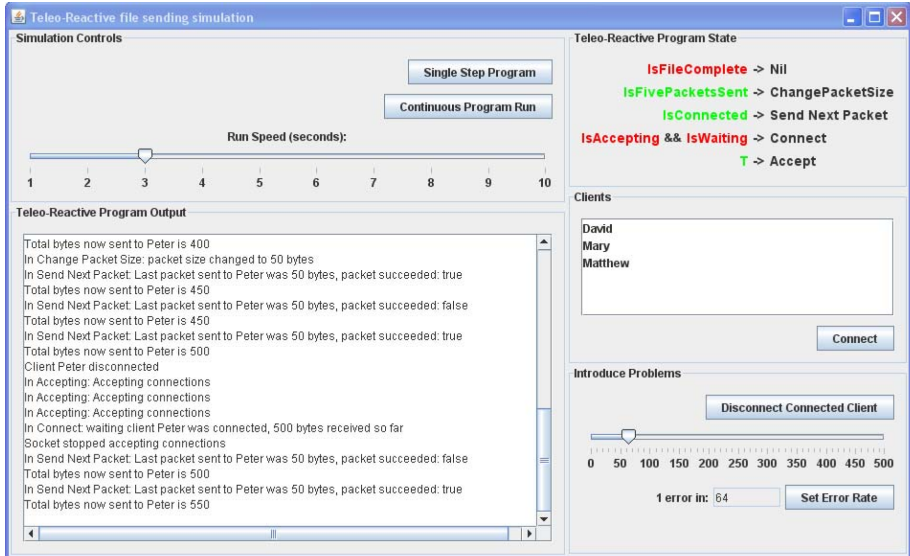
Conditionals contain the testCondition() method which must be implemented in extensions. This testCondition() method returns a boolean literal indicating whether the condition is true or false. A conditional also returns false when, during evaluation within the TRProg class, an error is encountered.

5 Example

The example presented here is a simple simulation of a file sending application presented from the point of view of the server side. The example extends the framework to simulate a simple protocol which we assume has little or no error handling abilities. The program contains several condition-action pairs leading to its main goal. For the sake of simplicity the file is always the same and is fixed length, with the aim being to send the file to the currently connected client. The example demonstrates how unexpected events such as random client disconnection and noisy communications are recovered from. This example is not designed as an accurate representation of networking components, it only serves to highlight some of the features and benefits of the approach. Figure 4(b) shows the example in mid-execution with one of the four clients connected.

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

(a) Alternative T-R state view



(b) 'True' conditions will be shown in green. 'False' conditions will be shown in red

Fig. 4. Screenshot from the example program

5.1 T-R Elements

The goal of the example is the same as the top level condition, to finish sending the file to the connected client. With the file sent the program does nothing until the top level condition turns false again. Figure 4(a) shows an alternative view of the 'Teleo-Reactive Program State' section from the example.

The conditions in the example are:

- **T** - As this is always true its action will always execute if no higher precedence condition is true.
- **IsAccepting** - The example simulates a socket which accepts incoming connection attempts. This condition returns the accepting state of the socket.
- **IsWaiting** - This condition reports whether there is a client in the waiting state.
- **IsConnected** - The client is connected after it is moved from waiting to the connected state.
- **IsFivePacketsSent** - Returns true when a multiple of 5 packets has been sent to the connected client.

- **IsFileComplete** - True once the whole file has been transferred to the connected client.

The actions in the example are:

- **Accept** - Simply turns the accepting state of the socket to on.
- **Connect** - Any waiting client is connected and the Accepting state is switched off.
- **Send Next Packet** - A packet is sent to the connected client at the current packet size.
- **Change Packet Size** - The packet size is adjusted using a simple algorithm based on statistics from the previous packet sending attempt.
- **Nil** - Performs no action.

5.2 Controls

The simulation GUI controls enable the user to set a run speed by changing the run speed slider and clicking the 'Continuous Program Run' button. This sets the simulation to perform a full pass through its action list every X seconds. The 'Single Step Program' button performs a single pass at each press and turns off automatic running.

The program state area gives visual feedback on the current progress in the T-R program. In the GUI a 'true' condition is coloured green, whilst a 'false' condition is coloured red. The action associated with the highest precedence 'true' condition is executed and will continue to execute at each pass while it remains so.

There are four possible clients that can be connected to the file sending service. Clicking on one of these four names followed by 'Connect' performs the connection attempt.

The 'Introduce Problems' controls are used to inject 'unexpected' conditions. This includes a random disconnection of a connected client and the introduction of degraded communication. The level of degradation is set with the slider and 'Set Error Rate' button.

5.3 Working through the Example

In this test, we make a connection and begin sending the file to this client in blocks. We demonstrate how the program copes with the unexpected events of disconnection and changing quality of service. This simple example is intended to illustrate the benefits and robustness of the approach.

When the example application is started the only 'true' condition is the first 'T' condition. On the first pass, the accept action will be executed, making the 'IsAccepting' condition 'true'. This will be the only executed action on subsequent passes until a higher precedence condition evaluates to 'true'. If we now connect one of the clients, the 'IsWaiting' condition becomes 'true'. The conditions for the action 'Connect' are now completely satisfied and takes precedence

over any lower actions. Its execution on the next pass will connect the waiting client, stop the socket from accepting connections, and cause 'IsConnected' to return 'true'. The service will continue to send packets to the client unless a higher precedence condition becomes 'true'.

There are a variety of ways in which the connection to the client can be lost. Clicking 'Disconnect Connected Client' causes the only 'true' condition to be 'T' again. The program will then work back to the point where the problem occurred and continue sending packets until the entire file is sent. The file send is resumed at the point where the last successful packet was sent. If another client connects before the fist client, they must wait until this client disconnects.

Every multiple of five packet sending attempts the service has the opportunity to adjust the packet size to suit the current communication quality and keep the service running at optimal levels. By changing the error rate in the simulation we can show how this works. The packet size will increase or decrease as appropriate. Once the 'IsFileComplete' condition is satisfied no other action will be performed and the goal is complete.

It was simple to add self-optimising behaviour to the program, which is demonstrated by the changing packet size program function. We had not intended to implement self-optimization within the first version of the example, but it turned out to be very simple to add this function. Using our framework, constructing the application was remarkably easy, and with future iterations of the framework (adding further robustness and inherent validation checking), this will be an easier task.

A more elaborate example scenario could be chosen with the opportunity to introduce a greater variety of problems and some of further features of the framework such as ease of reuse and recursion. However, the example is simple enough to demonstrate the main advantages which a more complex scenario might have blurred.

6 Future Work

A possible way to extend the framework is in use of policies to specify and possibly dynamically replace high level goals. A more advanced method of use of policies in T-R agent control is in [13] where situation graphs determine good policies for groups of cloned T-R agents. It is claimed that the use of situation graphs enables policies to be evaluated taking into account objective states and not just perceptions, yielding a high degree of discrimination.

We are interested in further applications of the framework in the development of autonomic and self managing systems [14,15] where the system needs to continually adjust its behaviour to suit its operating circumstances. It is far easier to describe these systems by their high level goals than by their actual behaviour at any given moment, and thus we suggest that T-R programs have great potential for this domain.

T-R programs also provide a lot of opportunity for automatic learning techniques and much of the work based on T-R programs has been addressing this

issue. For example [16,17]. Automatic learning of new goals and dynamic adaptation would reduce human reliance and this is another direction for future work on the framework we would be interested in pursuing.

In our view and from an autonomic viewpoint beneficial adaptations should, where possible, be transparent to the framework user. Automatic validation techniques could be incorporated to inform the programmer of the problem at the point where an action is added to the program, perhaps using simulation of T-R programs. This is a logical progression for the framework and would provide further guarantees about the level of robustness and reduce or eliminate the possibility of a dead/live-lock situation.

7 Conclusion

T-R programs were originally invented for the robotics domain and for agent control. This work shows how T-R programs can be effectively applied at a higher level and with many benefits over more traditional approaches. This method of programming focuses on goals as a driving factor and produces code more closely associated with a thought process. This in turn produces more natural behaviour and increases the potential for adaptability.

The T-R framework makes it simple to produce programs that use this technique as we demonstrated with our example scenario. T-R programs can form the main structure of a program with additional T-R programs called hierarchically. Or a T-R program could be used on a subsection where the program is started as a background asynchronous thread.

This initial work demonstrates the viability of the approach to programming high level software systems and we have illustrated how this approach contribute's to robust program design and self-healing capabilities. At this early stage in development the framework has been shown to be very effective in recovering from errors and using natural goal progression to improve system design. We believe that the complex models and architectures in some other self-healing approaches are not necessary and in many cases these methods are limited in the number and types of errors which can be handled.

References

1. Nilsson, N.J.: Teleo-reactive programs for agent control. Journal of Artificial Intelligence Research 1, 139–158 (1994)
2. Giorgini, P., Mylopoulos, J., Sebastiani, R.: Goal-oriented requirements analysis and reasoning in the tropos methodology. Engineering Applications of Artificial Intelligence 18(2), 159–171 (2005)
3. Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., Mylopoulos, J.: Tropos: An agent-oriented software development methodology. Autonomous Agents and Multi-Agent Systems 8(3), 203–236 (2004)
4. Castro, J., Kolp, M., Mylopoulos, J.: Towards requirements-driven information systems engineering: the tropos project. Inf. Syst. 27(6), 365–389 (2002)

5. Van Lamsweerde, A.: Goal-oriented requirements engineering: A guided tour. In: RE 2001: Proceedings of the Fifth IEEE International Symposium on Requirements Engineering, pp. 249–262. IEEE Computer Society, Washington (2001)
6. Fowler, M.: UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley Longman Publishing Co., Inc., Boston (2004)
7. Dashofy, E.M., van der Hoek, A., Taylor, R.N.: Towards architecture-based self-healing systems. In: WOSS 2002: Proceedings of the first workshop on Self-healing systems, pp. 21–26. ACM, New York (2002)
8. Garlan, D., Schmerl, B.: Model-based adaptation for self-healing systems. In: WOSS 2002: Proceedings of the first workshop on Self-healing systems, pp. 27–32. ACM, New York (2002)
9. Hassan, S., Mcsherry, D., Bustard, D.: Autonomic self healing and recovery informed by environment knowledge. Artif. Intell. Rev. 26, 89–101 (2006)
10. Nilsson, N.J.: Teleo-reactive programs web site `http://robotics.stanford.edu/users/nilsson/trweb/tr.html` (last accessed 2009)
11. Hayes, I.J.: Towards reasoning about teleo-reactive programs for robust real-time systems. In: SERENE 2008: Proceedings of the 2008 RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems, pp. 87–94. ACM, New York (2008)
12. Gordon, E., Logan, B.: Game over: You have been beaten by a GRUE. In: Fu, D., Henke, S., Orkin, J. (eds.) Challenges in Game Artificial Intelligence, Technical Report. Papers from the 2004 AAAI Workshop, pp. 16–21. AAAI Press, Menlo Park (2004)
13. Broda, K., Hogger, C.: Determining and verifying good policies for cloned teleo-reactive agents. Int. Journal of Computer Systems Science and Engineering 20(4), 249–258 (2005)
14. Kephart, J.: Research challenges of autonomic computing. In: International Conference on Software Engineering (ICSE), pp. 15–22. ACM, New York (2005)
15. Sterritt, R., Parashar, M., Tianfield, H., Unland, R.: A concise introduction to autonomic computing. Advanced Engineering Informatics 19(3), 181–187 (2005)
16. Kochenderfer, M.J.: Evolving hierarchical and recursive teleo-reactive programs through genetic programming. In: Ryan, C., Soule, T., Keijzer, M., Tsang, E.P.K., Poli, R., Costa, E. (eds.) EuroGP 2003. LNCS, vol. 2610, pp. 83–92. Springer, Heidelberg (2003)
17. Choi, D., Langley, P.: Learning teleoreactive logic programs from problem solving. In: Kramer, S., Pfahringer, B. (eds.) ILP 2005. LNCS (LNAI), vol. 3625, pp. 51–68. Springer, Heidelberg (2005)