Configuring Collaboration of Software Modules at Runtime

Willibald Krenn and **Franz Wotawa**
Graz University of Technology, Institute for Software Technology
Inffeldgasse 16B/II
A–8010 Graz, Austria
wkrenn, fwotawa @ist.tugraz.at

Abstract

We present an approach for (re-)configuring the collaboration of software modules on board an autonomous device. The proposed methodology largely is based on principles of logics: Different configurations are evaluated on the fly before one configuration is chosen and applied to the system. Each configuration has its own semantical meaning that is also included in the decision process. The set of all possible configurations is stored in a knowledge base that is queried before choosing a configuration. The presented approach allows to specify preferred configurations. We present first results obtained by running a prototype implementation of the presented methodology at the end of the paper.

Introduction[1]

In order to increase the over all durability of autonomous systems, these systems are often fitted with additional functional redundancy. This includes - whenever the device has some portion of control software - some set of software modules that control one particular part of the redundant functionality. In the simplest case, switching the software module configuration is done when some fault has been detected in one primary function and some redundant backup is needed in order to continue operation. In a more general approach, however, the system might be equipped with sensors, acquiring information about the environmental conditions the device has to operate in and the running configuration is selected depending on the union of detected faults and current environmental conditions. Of course, in order for this to be feasible, the device has to have knowledge about the configurations (whether they are pursuing similar goals, etc.) and there also has to be knowledge on how to interpret external sensor readings. In this paper we assume the system was given the knowledge about all valid configurations, on how to interpret sensor readings, and about the semantics of each configuration. We present a control algorithm that dynamically selects the best matching software configuration based on a history of past events, current (environmental) conditions, and preferences as specified by the system developer.

[1]This work has been supported by the FIT-IT research project *Self Properties in Autonomous Systems (SEPIAS)* which is funded by the Austrian Federal Ministry of Transport, Innovation and Technology, and the FFG.

As motivating example we present a container tracking unit with the duty to periodically transmit position data. The data transmission and the position acquisition functionality is available in several, different configurations reflecting the hardware redundancy. Although multiple configurations provide the same basic functionality, each of these configurations often has its own limitations regarding, e.g., precision of measurement, energy consumption, and others.

In the next section we present the motivating example and sketch our solution. Based upon insights derived from the example, we then compare the (informal) problem to solutions found by evolution. After these preliminaries, we formalize the problem and present our solution including some first results derived from a simple case study. After mentioning related research, we conclude.

Motivating Example

As already mentioned in the introduction, we use a container tracking unit (CTU) as our motivational example. The CTU thereby stands representatively for any autonomous device that features at least some hardware and software redundancy. In our case, the tracking unit has two different communication channels at its disposal: One GSM channel that is supposed to work most of the time, and one near-field channel that only works if there is another CTU in the neighborhood providing relay services, or some other terminating station that does the same. Apart from two different communication channels, the device has mainly two redundant ways of acquiring position information: The obvious one is by using the GPS service, another one is by identification of the GSM cell-ids and translating them to some geographical position. Of course, it may also be possible to ask another CTU in the neighborhood for help, using the GPS position of that device. Energy is provided by rechargeable batteries and a solar panel. The system is able to infer how much energy is available.

Each of the described hardware capabilities needs some piece of software on the main controller in order to be complete and functional. The problem we are facing now is that we have to select a configuration at run time that implements some intended functionality and considers the situation the device is in.

Example 1. *We give a textual example of a set of meaningful configurations for a CTU:*

Configuration 1: Send position information acquired by GPS with GSM.
Configuration 2: Send position information acquired by GPS with the near-field communication link (NF).
Configuration 3: Put the system into a low energy mode under certain conditions.

As can be seen, each of these configurations provide meaningful functionality when selected and applied to the system. Configuration one and two both send position information and differ only in the communication channels used. Thus, these configurations provide the same high-level functionality (attain the same goal) and are needed to implement found hardware redundancy. Configuration three, which puts the system into a low energy mode, may set the system in a state where configurations one and two are not applicable anymore, thus a configuration that reverses the effects from configuration three may be needed. The problem we face is to select configurations that best fit to changing environmental and internal conditions.

We give a more precise formulation of the problem later in the paper. In this section, we sketch how we intend solve the problem. Basically, we give the CTU a complete description of all valid configurations in form of a knowledge base. Note that we do not enumerate all configurations but instead describe in a high-level fashion how some configuration can be achieved. The device has to infer from this knowledge base all preconditions, postconditions and a sequence of software modules in order to build a valid configuration. It goes without saying that we also include the knowledge about what functionality the device shall go for inside this knowledge base. Apart from this pre-defined information, the device uses the knowledge base to rate each configuration according to some criterion. In our case the criterion will be fault absence. In other words, we let the system monitor (unexpected) faults that lead to not-working configuration instances. Not-working configurations are penalized in future searches of the knowledge base. However, because the autonomous system is under the influence of ever-changing environmental conditions, we allow the system to re-evaluate once found not-working configurations in order to check whether some blocking external event has disappeared over time.

The problem of selecting configurations, evaluating them, and applying them is quite common. It too can be found in bacteria that have to switch configurations, e.g., when they find food in order to start utilizing the food. When no food is present, the configuration has to switch again in order to turn off the production of chemicals that are used to "stomach" the food. It is an interesting fact that bacteria use some sort of knowledge base in combination with a sophisticated control mechanism to successfully solve the problem of selecting configurations. The employed control mechanism not only allows selecting a configuration but also controls the intensity of the reaction. This is similar to what we're searching for when we want to provide the user of our CTU (and the CTU itself) the freedom of specifying how often a configuration should be chosen over time.

The next section gives a short introduction to cell biology so we see how these configuration changes are made in cells. Thereafter, we formalize the problem and present an algorithm that draws upon concepts introduced in the next section.

Similar Systems in Nature [2]

"As researchers untangled the genetic code and the structure of genes in the 1950s and 60s, they began to see genes as a collection of plans, one plan for each protein. But genes do not produce their proteins all the time, suggesting that organisms can regulate gene expressions." (dnaftb.org)

In the following, we sketch the processes involved when producing proteins within a cell. At first some area of the DNA - some gene - is exprimed which means that the double helix is locally unwinded and separated. Inside this "bubble" of separated DNA, the DNA gets transcribed to RNA while the "bubble" wanders along the gene. At the end of the gene and after releasing the transcribed RNA, the "bubble" closes. The produced RNA is then used by some ribosome as plan for constructing a certain protein. Of course, and as already stated in the quotation at the beginning of this section, not all genes are always active. There exists some regulatory mechanism inside the cell that employs different mechanisms in order activate and/or deactivate a certain gene. A so called promoter (Wray *et al.* 2003) is used for activation or repression purposes. Found upstream the gene it controls,

"the promoter contains specific sequences that are recognized by proteins known as transcription factors. These factors bind to the promoter DNA sequences and the end result is the recruitment of RNA polymerase, the enzyme that synthesizes the RNA from the coding region of the gene. [...] Promoters represent critical elements that can work in concert with other regulatory regions (enhancers, silencers, [...]) to direct the level of transcription of a given gene." (wikipedia.org)

Without going into further details – we just want to mention that about 30 000 genes are apparently enough to "run" a human being while it needs 20 000 for a simple roundworm (ornl.gov) – we observe the following: DNA (the knowledge base, so to say) consists of genes. One gene encodes one bio-active entity, namely a protein. Gene activity is controlled by concentrations of certain proteins. A protein may influence the transcription of the gene that encodes itself. Each cell comprises a mechanism to transcribe the DNA and build up proteins.

What makes our approach similar to the working mechanism found in cells is the fact that we also have a knowledge base of all possible, good, configurations that attain certain goals. We also need an interpreter to read that knowledge base, extracting usable configurations. Lastly, we also face the challenge to regulate active configurations.

After presenting the preliminaries, we formalize the problem of configuration selection and present an algorithm that leans against ideas taken from transcriptional regulation.

[2]This section contains simplifications to some very high extend.

Formalization

Until now we have informally referred to configurations. The following definition introduces configurations formally.

Definition 1 (Configuration)**.** *A configuration is a tuple $(A, C, \gamma, G, activity, damping)$ where*

- *A is a sequence of literals describing software modules*
- *C is a sequence of sets of (pre-) conditions and $|C| = |A|$.*
- *$\gamma : \mathcal{R} \mapsto \mathcal{R}$ is a function mapping a real number to a real number. This function specifies an activity profile for the configuration and is used when calculating the weight of a configuration.*
- *G is a literal describing the goal of the configuration*
- *$activity \in \mathcal{R}$ is a real number describing the activity of the configuration*
- *$damping \in \mathcal{R}$ is a real number specifying a damping factor.*

In addition there exists an acceptance criterion for each configuration, which we assume is the same for all configurations within the system. Each configuration fulfills one particular goal when applied to the system. Software modules of a configuration may, of course, influence the weight of any configuration. In difference to a cell, that may exprime several genes at once, we sequentially select configurations and apply them because we manually have to invoke every software module. Therefore, we define a hyper-configuration.

Definition 2 (Hyper-Configuration)**.** *A hyper-configuration is created by applying a sequence of distinct configurations repeatedly to a system. Each configuration thereby is applied as long as it needs to (a) attain its goal or (b) to violate its acceptance criterion.*

Depending on the system, we have to find a sequence of hyper-configurations that best fit the current environmental and internal conditions. Without specifying the weight calculation yet, Algorithm 1 shows how we calculate a sequence of hyper-configurations. Seen over time, the system configures software modules in a way that a maximum amount of goals can be fulfilled.

The algorithm comprises four main parts: (1) At first internal values used for weight calculation are updated according to applied configurations. During this step, activity counters are aged, e.g., divided by two. Then, (2), weights are calculated and configurations are sorted according to that weight. Next, (3), all configurations that were applied too often (according to the weight), all configurations where the conditions are not fulfilled, and configurations where a higher rated alternative exists in the list of valid configurations are removed. (4) All remaining configurations are said to be valid ones and candidates to be applied. Therefore the system now loops through them, beginning at the most important one, applying each. After some amount of time the system has stayed in a configuration, the next one from the list is applied. In case a configuration does not satisfy the acceptance criterion, the damping factor is increased. Otherwise, the system will decrease the damping factor if not equal to zero. There are different ways the system may handle the activity counter: Two of them being incrementing the counter every time a configuration is applied, or, alternatively, incrementing the counter every time a configuration is applied and passes the acceptance test.

Algorithm 1 ApplyConfigurations

```
for all times do
  UpdateActivityAndDampingFactors();
  CalculateWeights();
  SortConfigurations();
  RemoveConfigurationsAppliedTooOften();
  for all configurations do
    while Conditions are not satisfiable do
      RemoveConfigurationFromList();
      SearchForHighestRatedAlternative();
    end while
    RemoveConfigurationFromList();
    RemoveAlternatives();
    ApplyConfigurationOnce();
    CheckAcceptance();
  end for
end for
```

For the presented motivational example, it makes most sense to take fault-absence during the run of each software module as acceptance criterion. The check can be implemented by some monitoring function that informs about discrepancies between expected result values (no error code, no timeout) and observed ones (error code, timeout).

After laying out the principal idea, we describe our algorithm in the following section.

Algorithm

While the sketched algorithm in the previous section very nicely shows the basic idea, it needs to have an enumeration of configurations as a working basis. Depending on how much different valid configurations exist, enumerating all of them may lead to a large amount of data with lots of unwanted duplication. Therefore, our implementation works on the basis of a knowledge base that stores all configurations in an implicit form. Informally, common subsequences of software modules within configurations are merged and stored only once. "Or" operators are used to separate configurations after common subsequences again and weights are used to guide the selection process.

Example 2. *Suppose $C_1 = a, b, c$ and $C_2 = a, b, d$ are two configurations attaining the same goal with a,b,c,d being software modules. We then can write all possible configurations as $(a \wedge b \wedge c) \vee (a \wedge b \wedge d)$ which simplifies to $a \wedge b \wedge (c \vee d)$.*

We use a language based on horn clauses to describe configurations inside our knowledge base:

Definition 3 (Configuration Knowledge Base)**.** *A configuration knowledge base contains all valid configurations for the device. The knowledge base is a tuple (P, R, G, γ) where*

- *P is a set of propositions.*

- *R is a set of horn-clauses of the form $x_1 \land \ldots \land x_n \rightarrow y$ where $x_1, \ldots, x_n, y \in P$. $x_1 \land \ldots \land x_n$ is the antecedence and y is the consequent.*
- *$G \subseteq P$ is a set of goals.*
- *$\gamma : P \times \mathcal{R} \mapsto \mathcal{R}$ is a function mapping a real number to a real number for one consequent in P.*

We omitted the functions returning activity and damping factors in the definition. Set P contains all conditions used by the configurations. The set R contains all configurations rewritten in form of rules. We also defined goals as subset of the conditions so rules can argue about goals. Note that function γ takes one more argument describing a rule consequence of a configuration.

The selection process of configurations as presented in the previous section works with a "globally best" strategy. Because of the merging of common software module sequences, we now use the strategy "locally best": Extracting one particular configuration from the knowledge base involves a guided search (starting by the highest rated goal) that uses backtracking techniques. The weights that we have introduced act as guidance when the system has to select between different branches in or-connected clauses. Therefore, the weight of one complete configuration as presented in the last section is broken down on rule level: Each consequent of a rule has to be connected to some weight. This changes the semantics of configuration selection, as we are searching for a local-best (at each or-decision) alternative now, instead of a global best as described in the last section. The advantage of this approach apart from saving CPU cycles is that similar configurations that share a common sequence of software modules will be rated similarly, regardless which of the configurations sharing the common sequence was applied. To put it differently, if two configurations share an sequence and one of them gets applied but fails to satisfy the acceptance criterion because of a fault, both configurations will be penalized.

The employed backtracking search algorithm excludes all configurations of one goal that are not satisfying the configuration conditions or that have been run too often. After finding a valid configuration and applying it, a new search is started in order to find a configuration satisfying another goal until all goals were processed. In the worst case – if no configuration satisfies its conditions – the proposed algorithm of course has to look at all stored configurations.

Weight Calculation

The central issue we yet have to present is the calculation of the weight. Weight has to reflect several things at once: (A) It has to indicate how successful it is to apply a configuration. (B) It has to reflect how often a particular configuration has already been applied. (C) It also needs to reflect preference criteria supplied by the system developer. We therefore propose a system that depends on activity profile functions that are supplied to the system. In addition two values are tracked by the system, namely activity and damping factors. Figure 1 sketches the idea behind. In our proposed system, weight is calculated as $S(a) * L(a) * (1 - D)$ where $S(a)$ is the slope of the supplied activity profile function (γ) at the point of the current activity, $L(a)$ is the distance of the point of current activity to the next local maximum of γ, and D is a damping factor that is tracked by the system and corresponds to the number of times a configuration has been applied but failed to achieve the acceptance criterion due to, e.g., unforeseen faults. The two values tracked by the system for each rule consequence are the activity factor which counts the number of times a certain configuration has been applied, and the damping factor (D) that counts the number of times the configuration was applied but did not achieve acceptance.

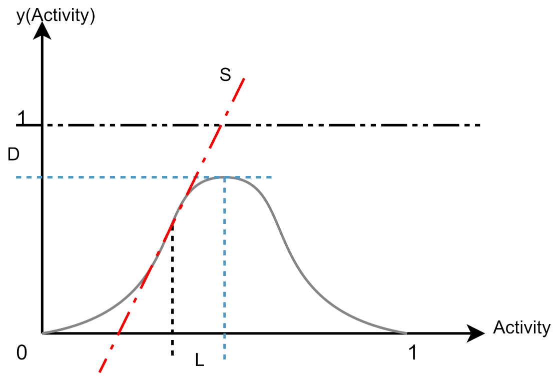

Figure 1: Calculating γ-based $Weight = S * L * (1 - D)$

The sole information the developer has to supply is the activity profile function γ. Depending on the needs of the system engineer, γ will look very different. In the easiest case, the function may degenerate to some straight line, in more complex cases, a function having more than one maximum may be supplied. In any case, it is possible for any software module to change γ at runtime, adding flexibility to the system by changing the preference criteria of configurations. The calculated weight is similar to a gradient, because the closer the activity of one configuration gets to the local maximum, the smaller the weight will be. In effect, the weight calculation function is constructed such that the activity of a configuration is brought to and held at some local maximum.

This ends the presentation of our algorithm. As already mentioned, we now give an example (created by hand) that demonstrates the workings of the presented algorithm.

Example 3. Figure 2 shows a graphical representation of following three configurations inside the knowledge base. For the sake of clarity, we omit everything but the sequence of software modules and goals and assume that all conditions for each configuration are always fulfilled.

C1: Goal: p; Sequence: ..., SendGSM, StoreTime
C2: Goal: p; Sequence: ..., SendNF, StoreTime
C3: Goal: p; Sequence: ..., Sleep

We use δ to denote a function that computes the weight by $S(a) * L(a) * (1 - D)$ where S calculates a normalized slope. The three configurations can be represented by fol-

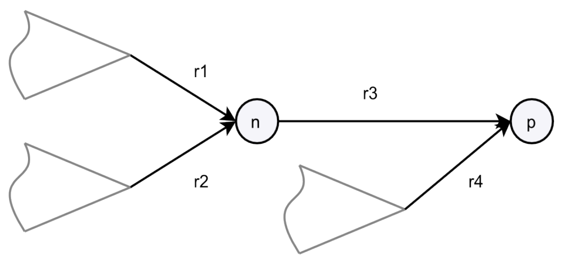

Figure 2: Sketch of rule set.

lowing rules, that match Figure 2.
(r1) $n \leftarrow \ldots \wedge module(\texttt{SendGSM})$
(r2) $n \leftarrow \ldots \wedge module(\texttt{SendNF})$
(r3) $p \leftarrow n \wedge module(\texttt{StoreTime})$
(r4) $p \leftarrow \ldots \wedge module(\texttt{Sleep})$
Rule four does not do anything meaningful in this example, except initiating a resting phase. It is mainly used to demonstrate that all configurations are selected by the system over time. The intention of the knowledge engineer is to slightly prefer configuration number one. Furthermore, the knowledge engineer has determined S for each rule, the decay value for the system, and increment values for damping and activity factors:
$\gamma_{max} = 1, \gamma(P, 0)'_n = 0.5, \gamma(P, 0.25)'_n = 0.25,$
$\gamma(P, 0.5)'_n = 0.5, \gamma(P, 0.75)'_n = 0.3,$
$\gamma(P, 0.875)'_n = 0.2$
$activity_{incr} = 1, damping_{incr} = 0.2$
$D_{r1} = 0, D_{r2} = 0.2, D_{r3} = 0, D_{r4} = 0.9$

We now calculate the weights of all configurations and show which of the configurations will be chosen by the system. Note that we increment the activity factor whenever a rule was selected. Alternatively, we could be more restrict and only increment activity factors from rules that led to working configurations. As all configurations have the same goal, the calculated hyper-configuration only consists of one entry.

1. $$\left.\begin{array}{l}\delta_{r1}(0.5, 1.0, 0.0) = 0.50\\ \delta_{r2}(0.5, 1.0, 0.2) = 0.40\\ \delta_{r3}(0.5, 1.0, 0.0) = 0.50\\ \delta_{r4}(0.5, 1.0, 0.9) = 0.05\end{array}\right\} \rightarrow \text{config. } 1 : \langle r3, r1\rangle \text{ sel.}$$
   We assume the configuration passes the acceptance criterion. The values of the activity factors are $activity_{r3} = 0.5, activity_{r1} = 0.5$.
   Following damping factors are not equal to zero: $D_{r2} = 0.2, D_{r4} = 0.9$

2. $$\left.\begin{array}{l}\delta_{r1}(0.5, 0.5, 0.0) = 0.25\\ \delta_{r2}(0.5, 1.0, 0.2) = 0.40\\ \delta_{r3}(0.5, 0.5, 0.0) = 0.25\\ \delta_{r4}(0.5, 1.0, 0.9) = 0.05\end{array}\right\} \rightarrow \text{config. } 2 : \langle r3, r2\rangle \text{ sel.}$$
   We assume the configuration does not satisfy the acceptance criterion because, e.g., a fault occurs. The values of the activity factors are $activity_{r3} = 0.75, activity_{r1} = 0.25, activity_{r2} = 0.5$. Following damping factors are not equal to zero: $D_{r2} = 0.4, D_{r3} = 0.2, D_{r4} = 0.9$

3. $$\left.\begin{array}{l}\delta_{r1}(0.25, 0.75, 0.0) = 0.19\\ \delta_{r2}(0.50, 0.50, 0.4) = 0.15\\ \delta_{r3}(0.30, 0.25, 0.2) = 0.06\\ \delta_{r4}(0.50, 1.00, 0.9) = 0.05\end{array}\right\} \rightarrow \text{config. } 1 : \langle r3, r1\rangle \text{ sel.}$$
   We assume the configuration passes the acceptance criterion. The values of the activity factors are $activity_{r3} = 0.875, activity_{r1} = 0.625, activity_{r2} = 0.25$. In addition, following damping factors are not equal to zero: $D_{r2} = 0.4, D_{r4} = 0.9$
   Note that r1 was chosen over r2 because of the damping factor.

4. $$\left.\begin{array}{l}\delta_{r3}(0.2, 0.125, 0.0) = 0.025\\ \delta_{r4}(0.5, 1.000, 0.9) = 0.050\end{array}\right\} \rightarrow \text{config. } 3 : \langle r4\rangle \text{ sel.}$$
   Finally, the system chooses configuration three. We want to emphasize again, that we have assumed that all conditions for each configuration could be satisfied all times.

This concludes the presentation of our algorithm. The next section gives first results obtained by working with a prototype implementation.

Experimental Results

Step	Faults	act. factor always inc.	act. fact. on success inc.	random
1	GSM	xC2	xC2	xC2 (1)
2	NF	xC4	xC4	C2 (1)
3	NF, GPS	C2 (pos. of step 1)	xC4	xC4 (2)
4	NF	xC4	C2 (pos. of step 1)	C2 (1)
5		C2 (pos. of step 4)	C2	C4 (2)
6	GPS, GSM	xC2	xC2	xC4 (2)
7	GPS, GSM	xC4	xC4	xC4 (2)
8	GPS	xC2	xC2	xC2 (1)
9	GPS	xC4	xC2	xC2 (1)
10	GPS, GSM	xC2	xC4	C5 (2)
11	GSM	C5	xC3	C5 (2)
12		C3	C5 (pos of step 11)	C4 (2)
13	NF	xC5	xC5	xC4 (2)
14	NF	C3 (pos. of step 13)	C3 (pos. of step 13)	C2 (1)
15	NF	C3	xC5	C2 (1)
16	NF, GPS	xC2	C3 (pos. of step 15)	xC2 (1)
17	NF, CO, GSM	xC4	xC3	xC2 (1)
18	NF, CO	C2 (pos. of step 17)	C2	C2 (1)
19	NF	C2	xC4	xC4 (2)
20	NF	C2	C2 (pos. of step 19)	C2 (1)

Table 1: Challenging test scenario for test purposes only.

We have implemented a first prototype of the presented algorithm on a Microchip PIC 18F micro controller. We have also created different knowledge bases containing rules for up to seven different goals for first experiments. A

Step	1	2	3	4	5	6	7
Fault	-	-	-	GSM	GSM	GSM	GSM
incr. always	C2	C4	C2	C4	xC2	C4	C4
incr. on reach	C2	C4	C2	C4	xC2	C4	C4

Table 2: More Realistic Scenario

knowledge base containing six goals that map to roughly eight different high-level configurations and 28 rules at interpretation level was used for the experiments described in this paper. In particular following configurations are stored in this knowledge base:

C1: book-keeping work (always part of a hyper-configuration, but omitted in the tables)
C2: get data by GPS, send it with GSM
C3: get data without using GPS, send it with GSM
C4: get data by GPS, send it with NF
C5: get data without using GPS, send it with NF
C6: disable gps
C7: enable gps
C8: shutdown system on low energy

Because of the limited processing power of the micro controller, we set γ as straight lines and no configuration is preferred: The maxima of the functions are shifted to the same value at high activity levels. This implements a strategy of using both configurations equally often. In addition to the weighted goal selection, we have implemented some experimental "x-from-y" strategy to decide between configurations two and three and between configurations four and five. From a total of 65536 bytes of program memory approximately two thirds are used for the search engine, a tiny operating system and the knowledge base. Besides program memory, a little bit more than 50% from a total of 3935 bytes RAM are needed.

Table 1 shows how our algorithm performs when confronted with a lot of configurations that violate their acceptance criterion (marked with an attached 'x' in the tables). Note that the test was constructed such that steps six to ten would fail in order to activate the four-in-eight strategy. Note also that the test prefers configuration two, but no strategy has been added to the knowledge base to reflect that fact. Steps 11 – 20 represent the most interesting part of the table. Because step 17 always must fail, our algorithm succeeds in seven out of nine cases. The random process also is quite successful but merely out of luck, as it chooses C2 in quite a long sequence. In reality we assume the faults to be longer lasting than shown in Table 1 and we assume some strategy (preference) for configurations in place. This will further improve the results. Table 2 shows – in that respect – a more realistic example, as we do not expect faults to pop up and disappear in high frequency.

Related Research

To our knowledge, the idea of applying principles from transcriptional regulation found in cells to a runtime - configuration problem is new. Other ways of solving this problem include, e.g., planning algorithms (Ghallab, Nau, & Traverso 2004). A very prominent example of this is the remote agent experiment (Williams *et al.* 1998) that was carried out on board the DeepSpace 1 probe in 1999. It used a planning system combined with a multi-layer execution engine, domain knowledge bases, and heuristics to derive a working configuration of sensors and actuators of the probe that could fulfill a given goal. In difference to our approach, the planner on board DeepSpace 1 did not rely on configurations as defined in this paper. The runtime configuration problem can also be solved by using decision trees (Console, Picardi, & Dupré 2003). While our approach is not as flexible as a planning system, we argue that it is more flexible than a normal decision tree approach, because our algorithm implicitly adds fault tolerance. TR-Programs (Nilsson 1994), like our approach, also have predefined sets of action sequences. Unlike the presented algorithm, however, TR-Programs do not allow dynamic weight changes.

Conclusion

We present the main idea and first results of a new way of selecting and applying software-module configurations. Because of the relatively small search space and the freedom to freely choose the complexity of γ the intended application area of the presented algorithm mainly comprises micro controllers that have low computing power. Except for valid configurations and preference criteria, the presented algorithm does not need any additional information, e.g., fault models, complex world models, or other kind of knowledge which underlines the intended target application.

References

Console, L.; Picardi, C.; and Dupré, D. T. 2003. Temporal decision trees: Model-based diagnosis of dynamic systems on-board. *Journal of Artificial Intelligence Research* (19):469–512.

dnaftb.org. Dna from the beginning, slide 33, genes can be turned on and off. http://www.dnaftb.org/dnaftb/33/concept/index.html.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning - theory and practice*. Morgan Kaufmann, Elsevier.

Nilsson, N. J. 1994. Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research* (1):139–158.

ornl.gov. Human genome project: How many genes are in the human genome?

wikipedia.org. Promoter, from wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Promoter.

Williams, B. C.; Muscettola, N.; Nayak, P. P.; and Pell, B. 1998. Remote agent: To boldly go where no AI system has gone before. *Artificial Intelligence* 103(1–2):5–47.

Wray, G. A.; Hahn, M. W.; Abouheif, E.; Balhoff, J. P.; Pizer, M.; Rockman, M. V.; and Romano, L. A. 2003. The evolution of transcriptional regulation in eukaryotes. *Molecular Biology And Evolution* 1377–1419.