

Scaleable Formalization of Imperfect Knowledge

R.Geoff. Dromey,
Software Quality Institute,
ARC Centre for Complex Systems
Griffith University,
Nathan, Brisbane, QLD 4111,
Australia.

“Building the right system, right”

1. Introduction

To realize the long-term goal of making verified software the norm we must first find practical ways of constructing formal specifications that accurately capture stakeholders’ requirements and intentions. One problem that stands directly in the way of this is the *imperfect knowledge problem* – where for many systems, subsets of the requirements are contributed by different stakeholders, with different assumptions, different understandings and different shades of vocabulary[1,2]. On top of this, members of the development team may also have different understandings and make different interpretations. The following table summarizes the different types of defects that may be found in a set of requirements as a consequence of imperfect knowledge.

Requirements - The Imperfect Knowledge Problem	
Problem	Cause of Problem
Inaccuracy	Misrepresentation of needs (MR)
	Misinterpretation of needs (MI)
Inadequacy	Misjudgment of needs (MJ)
Unnecessary	
Incompleteness	MR or MI or MJ
Inconsistency	MR
Redundancy	MR

Figure 1. Problems associated with requirements

Scale, where we have to deal with hundreds, if not thousands of requirements, takes the problem to another level of difficulty. Software engineers and formal methods specialists need to confront, and make substantial progress with this “double whammy” if the vision of verified to software is to be anything like a reality. Jones, et al, in their recent article about the “grand challenge” of verified software [3] talk about “resolving the problems of getting the specifications right” but go to remark that “... we can never bridge the formal/informal gap, ...”. Our position is that we are going to have to make substantial progress with getting to an unambiguous, accurate understanding of what to build. What we intend to show is that it is possible to substantially reduce the semantic gap between an informal statement of requirements and a formal, validated specification that unambiguously captures what is the “right” system to build.

2. Requirements Representation Issues

There are a number of ground rules and competing issues associated with the representation of the requirements for a system:

- Original statements of requirements will predominantly continue to be expressed in natural language for reasons of familiarity, expressive power and ease of expression. This means we are always going to have to start out with imperfect knowledge because ambiguity, inconsistency, redundancy and incompleteness are virtually unavoidable with anything substantial expressed in natural language.

- Expressing requirements in a formal language is probably the only effective method we have for resolving the imperfect knowledge problem.
- Expressing requirements formally has several knock-on effects. It introduces the risk that the formal representation will not accurately represent the original intention. This, in turn makes it essential that stakeholders validate any formal representation of requirements. For this to be possible and practical, stakeholders need to be able to understand the formal representation without the need for formal methods training. Clearly some existing formal representations while elegant and powerful are beyond the reach of untrained stakeholders.
- When scale and complexity are added to the equation it means that teams of developers and/or software engineers need to work collaboratively in parallel to formalize requirements and to undertake other associated development tasks in order to get projects completed in a realistic time. Most methods, formal or otherwise, do not have an obvious way of dividing up the work for teams and then constructively combining what individuals produce.
- Dealing with scale and complexity is not simply a matter of putting more people on a project – it has a cognitive dimension. Individuals have a limited short-term memory. We need to find a practical, scaleable way to overcome or compensate for this limitation.
- Expressing requirements formally is not enough; we need systematic and effective ways of assessing the suitability, adequacy, consistency and completeness of a set of requirements.

3. Principle of Design Economy

Here we present an outline of the strategy we are pursuing in response to these ground rules and competing issues. Results we have obtained over the last six years lead us to claim that the strategy shows promise as a contribution to the grand challenge of producing verified systems. Our overarching approach is to tackle the imperfect knowledge problem by seeking to use the *least information* and apply the *least effort* to build the right system, right, in the minimum time making best use of the given available resources. We call this strategy the *principle of design economy*.

This strategy implies a number of things:

- To build the *right system*, given that the initial requirements are informally expressed and therefore are likely contain ambiguities, aliases and other problems it will be necessary to *formalize* the requirements, find/correct defects and misinterpretations and preserve and clarify the original intention.
- The second essential for building the right system is for stakeholders to validate the formal representation of the requirements. For this to be possible *the formal representation(s) of requirements needs to be understandable by stakeholders* without their having had formal methods training. Retaining the original vocabulary (apart from the behavior-ordering information, aliases and other defects) is important to make validation practical.
- If developers and stakeholders are to cope with the complexity it will be essential that the representations and processes used avoid causing *short-term memory overflow*. Focusing on the detail in one requirement at a time (while not ignoring context) can significantly help with this.
- To be scaleable to large systems a method plus modeling language must, wherever possible, *allow multiple developers to work in parallel* and then formally combine what they have produced in a constructive way. It is even better if they can apply the same process (or same step in a process, e.g. requirements translation) to different inputs of the same type that need to be processed.
- The ability to build a system in *minimum time*, given the available resources, also depends heavily on multiple developers being able to work constructively in parallel.
- Using the *least information*, and employing the *least effort* to build the right system right, makes perfect sense – why would anyone choose to do otherwise? The challenge is to work out what we mean by the least information and the least effort. Our approach seeks to achieve this by making maximum use of information in the requirements – we build the system *out of its requirements*.

Here we will provide only an outline of the motivation, the strategy and tools we are evolving to tackle real-world complexity and construct systems in a rigorous way. Technical details and results achieved thus far formally, and by working with industry, using this whole approach, are discussed elsewhere [4-14].

4. Formalization on a Large Scale

If the goal is validated, verified, large-scale software-intensive systems, then our initial challenge is how to effectively and systematically use the tangle of requirements detail to advance towards a rigorous and unambiguous specification for the system that the initial requirements indicative. The ambiguity of natural language means that we have to find a practical way of formalizing and confirming what the requirements express if we are to preserve and clarify intention and find the inevitable defects in the original requirements. A useful initial goal should therefore be to cross the informal-formal barrier accurately without losing any useful intention in the original requirements. In contemplating this task, we cannot escape acknowledging and accommodating that individual software engineers only have very limited short-term memory – that is, they have no hope of keeping even tens, let alone hundreds of requirements in their consciousness at any one time. Given this limitation, formalizing a set of hundreds or even thousands of requirements seems like a daunting task. We may rightly ask is there any hope for establishing and maintaining intellectual control for very large systems?

What Herbert Simon said forty years ago about mathematics and problem-solving in his classic essay on “The Science of Design” both motivates and legitimizes a practical, formal way forward [15]. He said, “all mathematics exhibits in its conclusions only what is already implicit in its premises – all mathematical derivation can be viewed simply as change in representation, making evident what was previously true but obscure”. He then went on to say, “the ‘mathematical view’ can be extended to all problem-solving – solving a problem simply means representing it so as to make its solution transparent”. We can strengthen this strategy by utilizing the spirit of the *principle of information economy* from biology [16] which we can interpret as suggesting that “*we make maximum use of the minimum information (i.e., the requirements) to construct first a formal specification and then a design.*”

Translating Simon’s conjecture and the principle of information economy to software engineering suggests that perhaps the requirements for a system can be treated like the premises in a mathematical context. That is, the requirements are indicative of a formal specification for the system, if only we can find a suitable representation to make the transition while accommodating software engineers’ very limited short-term memory bandwidth. A way to do this is to frame the software engineering problem as a constructive, representation-changing task. For this we need to employ a representation (or set of representations) that will allow us to build a system *out of* its requirements which can enable us to achieve correctness by construction. If we are able to adopt such a constructive approach then formalizing a large set of requirements becomes a lot easier – we can formalize requirements one at a time. This allows us to focus on the detail in individual requirements without overflowing our short-term memory. It also gives us the best chance to preserve and clarify original intention in individual requirements because initially we do not have to concern ourselves with what other requirements contain (issues associated with the use of aliases in different requirements and inconsistencies can be picked up and resolved when the requirements information is formally integrated). The task becomes one of rigorous translation that seeks not to abstract, or refine, or ignore, but only to preserve the original meaning and the vocabulary (apart from the ordering information) used to express requirements in natural language – the aim being two-fold: to extract and formalize all the information in the original requirements that can be used subsequently throughout the development, and secondly, to carry out this task in a way that approaches *repeatability of construction* by different people, given the same initial information. Science and engineering have always built much of their reputation on the ability to do things repeatably – we should strive to do likewise.

To summarize, by formalization, our goal is to use rigorous translation then integration to make a meaning-preserving transition from a set of requirements expressed in natural language (which is inherently ambiguous) to one or more *formal integrated views* that reflect clarified and shared assumptions, shared understandings and a shared vocabulary. This is an absolute minimum precondition for creating a formalization that can both preserve and clarify intention and uncover defects in the original requirements.

5. Rigorous Translation

Our experience over the last six years, analyzing/modeling, and formalizing both small and large systems from a diverse set of application domains indicates that the information in individual requirements may be classified as *behavioral*, *compositional* or *structural* (this composition-structure-behavior ternary is not too different from what chemists use to deal with their molecular world). That is, when all the behavioral, compositional and structural information has been extracted from any individual requirement there is nothing useful left to formalize (non-functional requirements information maps as constraints or refinements onto one or more of the three categories). To preserve and clarify intention and to uncover and resolve defects in the original requirements three corresponding formal integrated views need to be constructed by processes of rigorous translation and integration. Below we show a set of simple requirements for a train station system which we will use throughout to illustrate requirements translation and integration using behavior trees and composition trees.

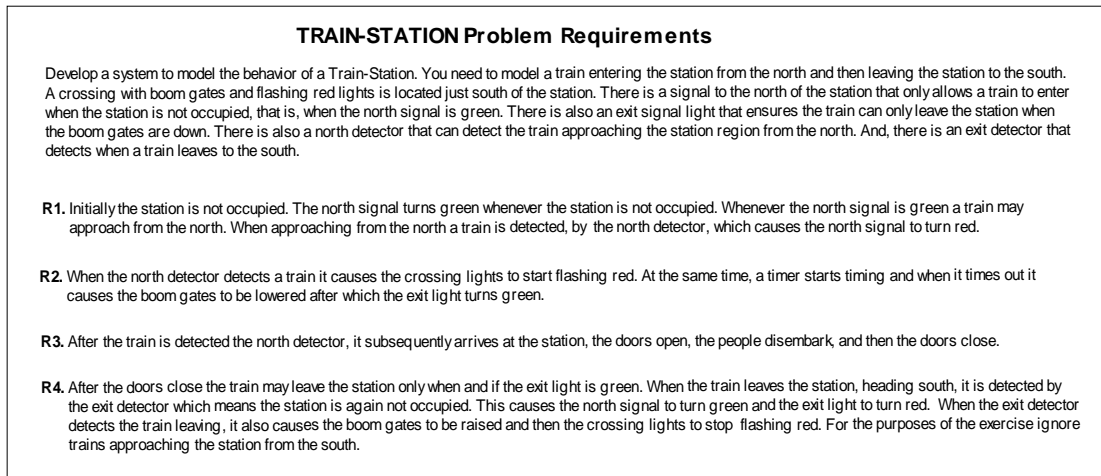


Figure 2. Requirements for Train Station Problem

5.1 Behavior Trees

Behavior Trees, [4,5] a graphical representation, which has been given a formal semantics based on a new process algebra, [6] allows us to formalize and integrate all the fragments of behavior expressed in individual requirements. Importantly, because requirements translation retains the original component/behavior vocabulary, and because behavior trees (BT) have a simple syntax and clear semantics, the formal representation of requirements is easy for stakeholders to validate without needing a formal methods background (by retaining the original vocabulary, behavior trees allow us to significantly reduce the *semantic gap* between the informal and formal representations). It is essential for stakeholders to validate the formal representation if we are to maximize the chances of preserving intention. (To date, the largest real-world system we have formalized with Behavior Trees had more than one thousand requirements). Translating a set of requirements to their corresponding set of formal behavior trees can be parallelized by allocating non-overlapping subsets of requirements to different developers. With appropriate tool support, that makes the dynamically evolving vocabulary available to all requirements translators, there is the potential to get a gain in speed of construction proportional to the number of translators involved – assuming they all do requirements translations at the same rate. This contributes to our earlier sub-goal of building a system in minimum time making best use of the given available resources. The result of translating requirement R1 in figure 2 to a requirements behavior tree (RBT) is shown below in Figure 3. In a behavior tree the $?? \dots ??$ indicates an event for a component, e.g., that the north detector has detected a train and $[\dots]$ indicates state realization by a component. A condition/decision for a component is indicated by $? \dots ?$ and arrows indicate sequential composition of behavior. More than one arrow emanating from a component-state node indicates concurrent behavior.

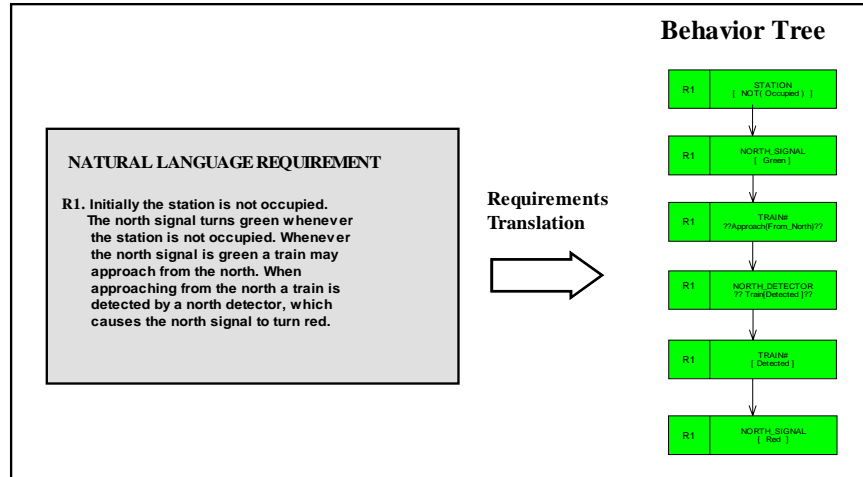


Figure 3. Translation of a requirement (R1) to a requirements behavior tree (RBT).

5.2 Composition Trees

Composition trees allow us to formalize and integrate graphically all the fragments of composition expressed in individual requirements. A composition tree is a vehicle for defining and representing the *system composition* - a unique structured composite of properties (including the complete vocabulary) of any given set of requirements, whether they imply a database system, a reactive system, a transactions-based system or some other type of system. We construct an integrated composition tree (ICT) incrementally by rigorous translation and integration of the composition fragments (the RCT) in each individual requirement. In figure 4 we show the composition tree that results from translating and composing all the compositional fragments in R1. Like the behavior tree formalization strategy, this process allows us to cope with complexity and detail in individual requirements without overflowing our short-term memory. Again the same vocabulary is retained which makes validation by stakeholders straightforward. Interestingly, a composition tree includes all the compositional elements and vocabulary in a behavior tree, except that in this instance they are structured to constructively support the design, implementation and defect detection for individual components.

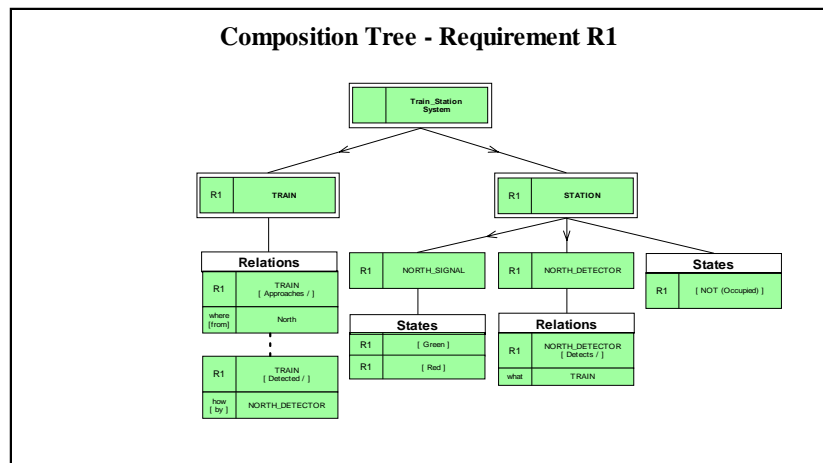


Figure 4. Translation of a requirement (R1) to a requirements composition tree (RCT).

While this duplication of information in a different form seems to go against our quest to use the least information and the least effort to construct a design we argue that this is not the case for the following reasons. Translating a set of requirements to their corresponding set of formal composition trees can be

parallelized by allocating non-overlapping subsets of requirements to different developers. This allows us to make maximum use of our available resources in constructing the individual composition trees. In fact, with tool support, composition tree translation can and should be overlapped with the BT translation.

5.3 Integration of Requirements Composition Trees

The process of integrating composition trees is straightforward. All information about each component, the states it realizes, the data it receives, the data it outputs, the relations it forms, the data it encapsulates, etc are all classified and integrated under each component. If a component is a system in its own right (e.g. the “Station” is a system in our example) then the components that make up that system are child nodes of that system component. Using this strategy the integrated composition tree represents the components of the system as a system-of-system hierarchy. The result of translating the compositional information in requirement R2 and then integrating it with the RCT for requirement R1 is shown below in figure 5. The complete integrated composition tree (ICT) that results from translating and integrating the composition trees for all the requirements in fig. 2 is shown in figure 8.

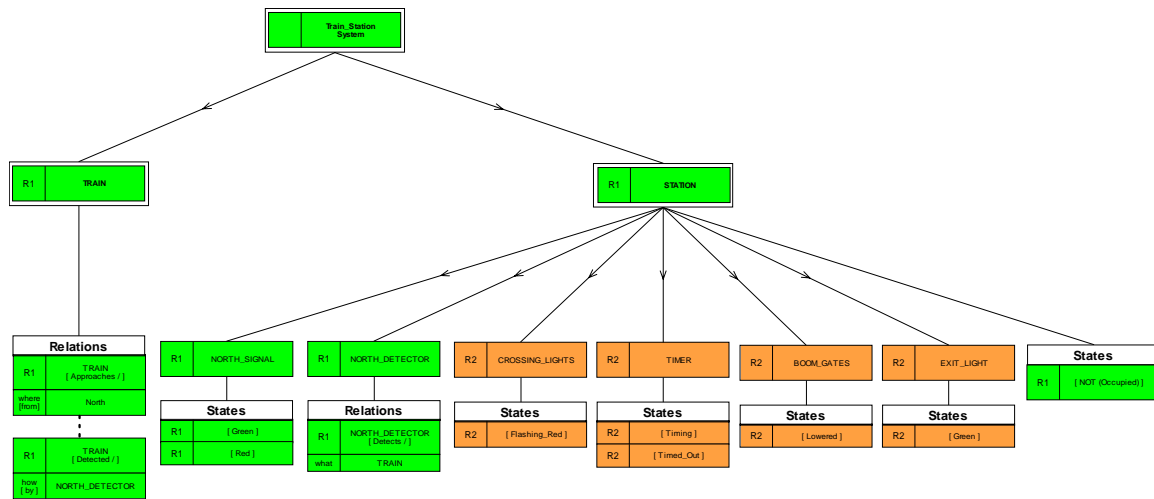


Figure 5. Integration of requirements composition trees for requirements R1 and R2 from Fig. 2.

The complete integrated composition tree for the original requirements serves six important purposes: (1) It comprehensively formalizes, structures and organizes all aspects of the vocabulary of a large system in a way that is constructively useful for subsequent steps in the development of the system. In other words, it is the vital tool we use to overcome the imperfect knowledge problem we raised at the start of our discussion (2) It organizes in one place, for each component (some of which are systems themselves) in a system, all the information about that component – this information tends to be widely scattered across the set of requirements. Having a comprehensive integrated view of all information about each component is vital to its subsequent efficient and accurate design and implementation. Building a comprehensive understanding of the components in a system also makes an important contribution to understanding the system. (3) It formalizes and integrates all data requirements information and constraints present in a set of requirements (4) It records the system-of-systems component hierarchy of a system. This structure defines the dominant shape and component multiplicity of the composition tree – it shows how each component is related to other components in the system (5) By using a process of integration it serves to identify a range of defects that are otherwise very difficult to detect when we have to deal with large numbers of requirements. (6) As we progress through the specification, design and implementation phases the composition tree can be refined as needed to incorporate design and implementation compositional information. By the time we have finished constructing a composition tree of the requirements we should already have a clear understanding of what type of system the requirements imply and a useful estimate of the size and relative complexity of the system. Once constructed, a composition tree becomes a vital asset to support evolution/change. It can also serve as a vehicle to quickly bring new team members up to speed on a project.

Class diagrams, ER-diagrams and information models have been used by other methods to capture some of the information that ends up in an integrated composition tree for a set of requirements. However, a composition tree has a number of important advantages over these static representations. (1) It is a tree rather than a network which makes it easier to understand and scale to handle real-world systems-of-systems. (2) Because a composition tree [13] is defined as a unique composite property of a set of requirements (it includes such attributes as the set of all components mentioned in the requirements, all data each component encapsulates, all the inputs a component receives, and so on) it approaches repeatability of construction by different people. (3) A team, using an appropriate cooperative editing tool, can potentially work constructively in parallel to translate and integrate the composition tree information derived from each requirement. This means, as for behavior trees, that with tool support, it is possible to get a gain in speed of construction that is proportional to the number of people involved – assuming they all do translations at the same rate. (4) With tool support for large composition trees it is also possible to look at different partial views. For example, we can choose to display a partial view of a composition tree that just shows all inputs and outputs for each component or all data encapsulated by each component, or all information for just a single component, and so on. In many system design situations an integrated view of behavior and composition is all we need to support the construction of a system design. However, there are other contexts where an integrated view of a structure, or set of structures that behavior takes place on, is also important.

6. Integrated Views of Requirements

Once a set of requirements have been formalized using behavior trees (and an integrated composition tree has been constructed to record the complete structured vocabulary for the requirements) and aliases have been resolved and removed and translation errors (due to ambiguity) have been identified by validation and correction of the formalized behavior trees and integrated composition tree we can say a set of requirements are in *First Normal Form*. While this normalization may not have the precision of some such frameworks it represents an important first step of a corrigible ladder of formalization that we can impose on a set of requirements for a system as we progressively refine the set towards a formal specification.

6.1 Integrated Behavior Tree

What is interesting and useful about the behavior tree representation, is that it supports composition of individual requirements one at a time into an integrated, component-state-based graphical view which is also tree-like in form. Informally, integration of two requirements represented as behavior trees proceeds by finding where the root node of one tree occurs in another and joining the two trees at that node (see for example figure 6). Formal details of how to do requirements integration are provided elsewhere [4,5]

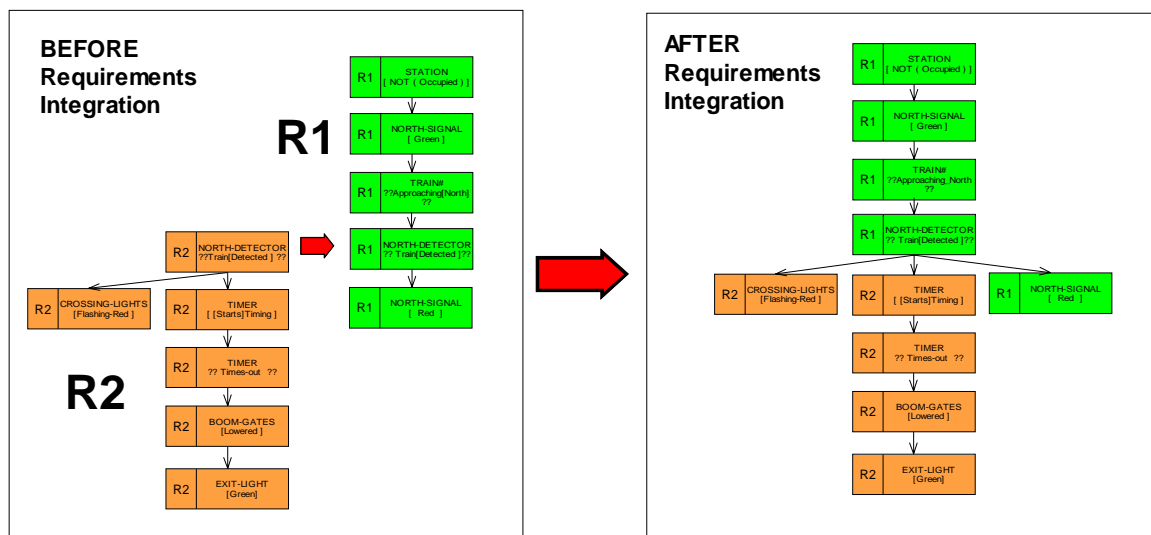


Figure 6. Integration of behavior trees for requirements R1 and R2 from Fig. 2.

By integrating a set of requirements we put each individual requirement in the broader behavioral context where it needs to exhibit its specific behavior. Only by doing this are we in a position to rigorously assess the suitability and contribution of each individual requirement to the integrated behavior of the system. Moreover, the integrated graphical view facilitates inspection and model-checking [14,17] of the integrated behavior of the system and the early detection and removal of completeness and other defects [11]. A formal integrated view of all the requirements constitutes *second normal form*. We call this integrated view, the *integrated behavior tree* (IBT) for a set of requirements. The complete IBT for the requirements in fig.2 are shown in fig.9.

Jigsaw puzzles give us some intuition of how and why behavior trees and the genetic design process they imply, works[10]. By some measures, composing a 1000-piece jigsaw puzzle, randomly scattered on a table, might be regarded as a daunting problem. However, we break the task down into a very simple process that we apply over and over. The process involves considering and putting in its appropriate place **only one** jigsaw puzzle piece at a time. The focus is upon where (the position) to put each piece. Interestingly, the *order* in which we put the pieces together turns out not to be important. Also, it is possible for more than one person to work on the problem. All this is possible because jigsaw puzzle pieces, as a set, **contain enough information to allow their composition** – a set of genes also has this property. Exactly the same properties apply for a set of functional requirements represented as behavior trees. Integration of requirements (which can be tool-assisted) is possible because the behavior encapsulated in each requirement has a precondition that must be satisfied in order for its behavior to execute (very often, the root of the behavior tree is this precondition). For example, for a light to turn on when someone flicks its switch, the power-grid must be delivering electricity to the switch (it establishes the precondition). For a requirement to be part of a system at least one other requirement in the set must establish the precondition it needs to execute – this defines the point of integration/composition for the two requirements. When we find requirements that cannot be integrated, it immediately points to incompleteness or other problems that need to be resolved. Forming a completely integrated view of a set of requirements pays due acknowledgement to the principle of totality – it establishes all the interactions among a set of requirements. An integrated behavior tree is usually only a tree in form – leaf nodes typically revert “^” (loop) back to an ancestor node and thereby specify repetition of behavior that was encountered earlier.

Refinement of the integrated view yields a behavior design in which the requirements are embedded. Building a system 'out of' its requirements (which are specifically labeled/tagged in the integrated view) in this way also simplifies the validation process and makes it easier to check and guarantee that stakeholders' needs are met and carried through to the design and implementation. And, most importantly, because we only have to deal with one requirement at a time, we gain much greater control over complexity and change[8]. While focusing on constructing an integrated view of all the behavior in a set of requirements is central to creating a design for any system, in general it needs to be supported by constructing an integrated view of the compositional information in a set of requirements.

6.2 Structure Trees

With some large systems much of the behavior takes place on a complex infrastructure – for example, a rail network, or a communications network. As a consequence, when the requirements for such systems are stated they may contain information including constraints about the structure that behavior is to take place on. While the compositional information about such structures fits appropriately in a composition tree, if we only capture the compositional information we will have left important information about such structures unformalized by the translation process. Structure trees allow us to formalize the structural information in requirements. We can construct a structure tree using a similar process to the one used for behavior trees and composition trees - we translate and integrate fragments of structural information spread across a set of requirements to create a structure tree. We could probably use a textual notation like EBNF to record the set of grammar rules for such structures. Our preference is however to use the integrated tree-like graphical form because it is easier to comprehend as a whole and integration helps find problems with the description of complex structures (which are often recursive). Jackson showed a long time ago with transaction-based data [18] the value of formally defining the structure of the data graphically then using the principle of correspondence to design the control structure for a program to process that data. In a

similar way, we suggest it is essential to align the behavior flow defined by the functional requirements with the structure that it takes place on. Having a formally defined structure tree allows us to do this.

7. Processes and Operations on Integrated Views

Once we have formal, tree-like, graphical integrated views of all the behavioral, compositional and structural information in the original requirements we are in a strong position to constructively and efficiently use this information to construct a system specification and then a system design.

7.1 Formal Integrated Specification Construction

To construct a system specification from an integrated view of the requirements we should only seek to make refinements/corrections that make the representation *accurate*, and *adequate* as far as stakeholders determine by validation using inspection. All that remains to finalize the form of the system specification is to make it *logically complete, consistent, non-redundant and completely necessary*. For this we can use inspection and tool support to implement a set of well-defined consistency and completeness checks for missing events, conditions and reversions (loop-backs to an ancestor node) that are easy to detect in an integrated behavioral view. For example, in fig. 9 we see near the bottom of the tree (as part of the R4 fragment) EXIT_LIGHT[Red] emanates from the STATION node. To be consistent this same behavior should also emanate from the root of the tree – it is missing and represents a defect. To turn this tree into a specification we need to remove the redundancy contributed by R4 and have the STATION node at the bottom of the tree revert “^” to the root node at the top of the tree. We also need to initialize all components at the top of the tree. When these steps are done we have a formal specification that has been constructed from a formal integrated view of the requirements – that is, the labeled requirements are embedded in the specification. A representation with these properties is in *third normal form*. We call this integrated view, that serves as the derived formal specification the *model behavior tree* (MBT) for a set of requirements.

7.2 Model-Checking and Simulation

Once we have a formal specification for a system we can take advantage of the underpinning process algebra formal semantics [6] of Behavior Trees to do simulation and model-checking. We have built a tool that converts graphically produced behavior trees to an input that can be executed by a simulator written in Mercury that is based on the process algebra. Our goal in the future is to extend this work so we can conduct large-scale distributed simulations using HLA-supported tools. We have also developed a facility that converts the graphical input to an Action Systems representation which is then directed to the SAL model-checker [17]. We can use this system to verify invariants and safety properties of the system [12,14]. Once we have conducted the simulations and model-checking and made any necessary corrections to the formal specification that preserve the properties of the first three normal forms we can proceed with refinements to the formal specification, where necessary, to produce a design.

7.3 Refinement of the Specification to Create a Design

Using an integrated behavior tree specification (the MBT) to construct a design implies at the highest level a clearly defined strategy. Our overall goal is to construct a design which is an integrated view that has the specification embedded. That is formally, the design, represented by a design behavior tree (DBT) which is logically stronger, implies the specification. The focus in making refinements is to add components and behaviors that concretize what the abstract state transitions of the specification require. The following example in fig.7 illustrates this point.

To formally support the refinement of a specification into a design it would be possible to devise a refinement calculus along similar lines to that devised by Carroll Morgan. What is fundamental about all refinements is that they must preserve and satisfy all constraints of the first three normal forms. In other words they must not violate but only strengthen the MBT specification. As it should be, what constitutes a valid design refinement that will satisfy the specification is in the hands of the designs and domain experts – the method and representation can provide no guidance on this other than any refinement must confirm to the constraints of the first three normal forms. For example, whether a designer chooses to use a hardware timer component to implement a timing event or whether the decision is made to implement the timer in software is clearly at the discretion of the designer.

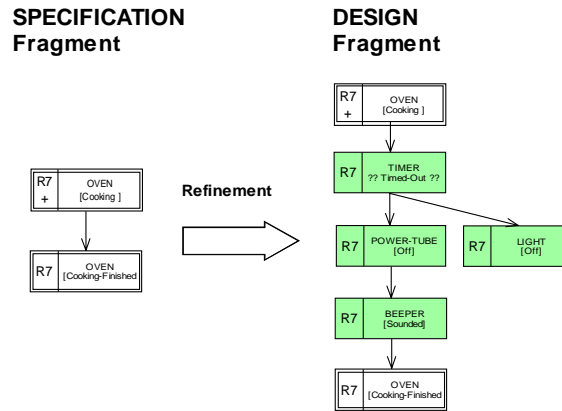


Figure 7. Refining a specification fragment into a design fragment.

8. Discussion

With the proposals sketched in this discussion paper it is important to ask how they can contribute to the long-term goal of producing large-scale software-intensive systems that are error-free? We will try to answer this question in two ways: (1) by responding to the six “big issues” Anthony Hall suggested need to be addressed to accomplish “the real Grand Challenge for formal methods ... to make correctness by construction the mainstream approach to software development.” [2] (2) by giving a brief qualitative assessment of the industry trials on large systems we have done over the last four years

8.1 Hall’s Six Big Issues - Or Correctness By Construction

Anthony Hall, practitioner and advocate of formal methods, identifies six “big issues” that he believes the specification and verification communities need to address. We will make a number of claims and comments about how our method, representations and tools address these issues. Independent validation of these claims needs to be done. This is something we are keen to work with other laboratories to do. We will now look at each of the six issues in turn.

1. *Early Requirements.* Hall asks how can we add rigor to use cases, scenarios, etc, which are essential for dealing with stakeholders? Behavior trees and Composition Trees directly address this problem. They have a simple formal semantics and syntax, and they expressly retain the vocabulary of the original requirements. The integrated requirements representations are formal and therefore unambiguous while still easily accessible to all stakeholders.
2. *Specification Languages.* Here Hall asks how can we have expressive specification languages and at the same time tractable proofs. We have built a tool to directly convert graphical Behavior Trees to the Actions Systems language which allows us to use the SAL model-checker to prove various theorems expressed in LTL. We have also built a simulator based on the formal semantics for Behavior trees which accepts as input the converted symbolic representation of graphically represented behavior trees. These tools allow extensive behavioral checking of a set of requirements that represent a specification and the refined integrated version (with the requirements embedded) that represents the design. The same representations are used throughout for analysis, specification and design. This simplifies the transition from one phase to the next. Analysis and formalization yield an integrated view of the requirements which is refined as necessary to create a design.
3. *Design Notations.* Hall asks how can we express the multiple dimensions of a design? We claim the combination of behavior trees, composition trees and structure trees with the transformations they allow are powerful enough to capture and accommodate the multiple dimensions of design – from databases, concurrent systems, through to systems of systems design.

4. *Concurrency. Here Hall asks can we express concurrency properties in a compositional way?* In the behavior tree representation of a system concurrent segments of behavior are manifested as sub-trees (with leaf-node feedback) at various levels of the overall integrated tree. Typically these concurrent segments function as communicating sequential processes. This produces a “compositional” representation of concurrency.
5. *Testing. How can we develop efficient test cases from our requirements?* Because behavior trees focus on building a system out of its requirements, it makes the task of validating requirements by inspection, simulation and testing a lot more straightforward. Also, because the behavior of all requirements is integrated it is straightforward to design a set of test cases to get complete behavior coverage of the tree and the requirements.
6. *Proof. How can we choose what to prove? How can we make proof accessible? How can we use proof for finding errors?* Choosing what to prove is always difficult. We have primarily used the graphic tools we have developed in combination with model-checking to check a range of safety properties. We are currently extending our tools to be able to check security properties as well. Going directly from the graphics to the model checking is a step towards making proof more accessible. We claim that the use of requirements translation, requirements integration, inspection and model-checking represent a powerful set of strategies for finding requirements defects. In particular integration of requirements is very powerful for finding incompleteness defects associated with a set of requirements. Many defects are detected by construction.

8. 2 Progress With Scale-up to Large Systems

We have used the techniques described to analyze the requirements, sometimes partially and sometimes fully a variety of medium and large-scale systems. The largest real-world system we have completely formalized with Behavior Trees and composition trees had more than one thousand requirements. The analysis was able to discover over 100 significant defects. When the integrated behavior tree for this system is printed on a very large printer it is approximately four meters wide and 1.5m deep. The composition tree is much smaller. Clearly, for systems of this size it is desirable to have a sophisticated multi-user tool.

At this stage the work we have done on large systems has only been done by either two or three people without cooperative editing tool support (we are currently adapting a version of our tool so that it will allow a team of people to concurrently edit the same document and at all times to see the impact of the work of other people on their work). In three of the large systems that we have worked on prior to the systems being built we have found a much higher serious defect rate than has been found by more conventional inspection and analysis methods [11]. In one case, as a result of our analysis they went and rewrote their requirements.

Overall, in qualitative terms the industry trials we have done suggest that the method is scaleable in terms of being able to create integrated views of the requirements of large systems. However multi-user tools support with very good graphics presentation capability, will be essential to make the method practical for use by large teams. At this stage we are able to do simulation and model-checking without any trouble on small systems. We are hoping to scale the use of these techniques to handle the much larger systems we have been analyzing. We have recently been approached by a large multi-national systems integration company to conduct significant trials of the method. Unfortunately most of the trials we have done on large systems have either been subject to commercial-in-confidence or other restrictions. We do however have available a small case study for a satellite control system that has a 23-page functional requirements specification. This is probably of sufficient size to give a feel for how the method works and its effectiveness.

9. Future Work

We are currently adding collaborative working capability to the new formal graphic modelling tools we have developed to support the proposed method [9]. This capability will allow us to experimentally measure and confirm the productivity increases that we predict are possible for teams using the method. Once we have the tool we also intend to conduct experiments regarding the finding of defects by teams.

Another challenge we need to address is how best can teams of designers navigate and work with very large integrated views of behavior models. We also need to scale up our model-checking capability.

10. Acknowledgements

This work is supported by the Australian Research Council – ARC Centre for Complex Systems.

11. References

- [1] Shlaer, S., S.J.Mellor, *Object-Oriented Systems Analysis: Modeling the World in Data*, Yourdon Press/Prentice-Hall, Englewood Cliffs, N.J., 1988, pp. 4-5.
- [2] A.Hall, Software Verification and Software Engineering: A Practitioner's Perspective, IFIP Working Conference on Verified Software (VSTTE 2006: <http://vstte.ethz.ch/>)
- [3] C.Jones, P.O'Hearn, J.Woodcock, Verified Software: A Grand Challenge, *IEEE Computer*, April. 2006, pp.93-95.
- [4] R.G.Dromey, From Requirements to Design: Formalizing the Key Steps, (Keynote Address), SEFM-2003, *IEEE International Conference on Software Engineering and Formal Methods*, Brisbane, Sept. 2003, pp.2-11.
- [5] R.G.Dromey, "Formalizing the Transition from Requirements to Design", in *Mathematical Frameworks for Component Software – Models for Analysis and Synthesis*, Jifeng He, and Zhiming Liu (Eds.), World Scientific Series on Component-Based Development, 2006, pp. 156-187.
- [6] R.G.Dromey, "Climbing Over the 'No Silver Bullet' Brick Wall", *IEEE Software*, Vol. 23, No. 2, pp.118-120, 2006.
- [7] R.Colvin, I.J.Hayes, A Semantics for Behavior Trees, 2006, <http://www.itee.uq.edu.au/~robert/DCCS/semantics/main.pdf>
- [8] L. Wen, R.G.Dromey, From Requirements Change to Design Change: A Formal Path, SEFM-2004, *IEEE International Conference on Software Engineering and Formal Methods*, Eds., J.R. Cuellar, Z. Liu, Beijing, September 2004, pp. 104-113,.
- [9] K.Lin, D.Chen, C.Sun, R.G.Dromey, Tree Structure Maintenance in a Collaborative Genetic Software Engineering System, in Proceedings of the Sixth International Workshop on Collaborative Editing Systems, November, 2004.
- [10] R.G.Dromey, "Genetic Design: Amplifying Our Ability to Deal With Requirements Complexity", in S.Leue, and T.J. Systra, Scenarios, *Lecture Notes in Computer Science*, LNCS 3466, 2005, pp. 95 – 108.
- [11] R.G.Dromey, D.Powell, Early Requirements Defect Detection, *TickIT Journal*, 4Q05, 2005, pp. 3 – 13,.
- [12] S.Zafar, and R.G.Dromey, Integrating Safety and Security Requirements into Design of an Embedded System. *Asia-Pacific Software Engineering Conference*, Taipei, Taiwan. IEEE Computer Society Press, 2005, pp.629-636
- [13] R.G.Dromey, System Composition: Constructive Support for the Analysis and Design of Large Systems, SETE-2005, *Systems Engineering/Test and Evaluation Conference*, Brisbane, Australia, 2005.
- [14] L.Grunske, P.Lindsay, N.Yatapanage, K.Winter, An Automated Failure and Effect Analysis Based on High-Level Design Specification with Behavior Trees, *Fifth International Conference on Integrated Formal Methods (IFM-2005)*, Eindhoven, The Netherlands, 2005.
- [15] H.Simon, "The Science of Design" in *The Sciences of the Artificial*, 2nd Ed., MIT Press, Cambridge Mass., 1990.
- [16] Loewenstein, W.R., *The Touchstone of Life*, Penguin, London, 1999.
- [17] The SAL Group. The SAL Intermediate Language. Tech. Report, UC Berkeley, SRI, Stanford University, 1999.
- [18] M.A. Jackson, *Principles of Program Design*, Academic Press, London, 1975.

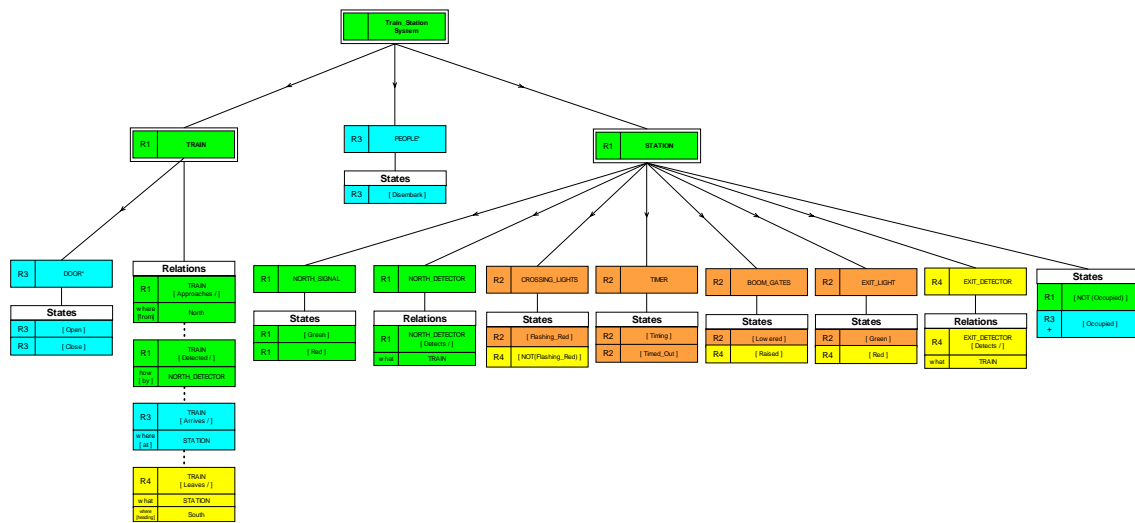


Figure 8. Integrated composition tree (ICT) for all requirements of Train Station System.

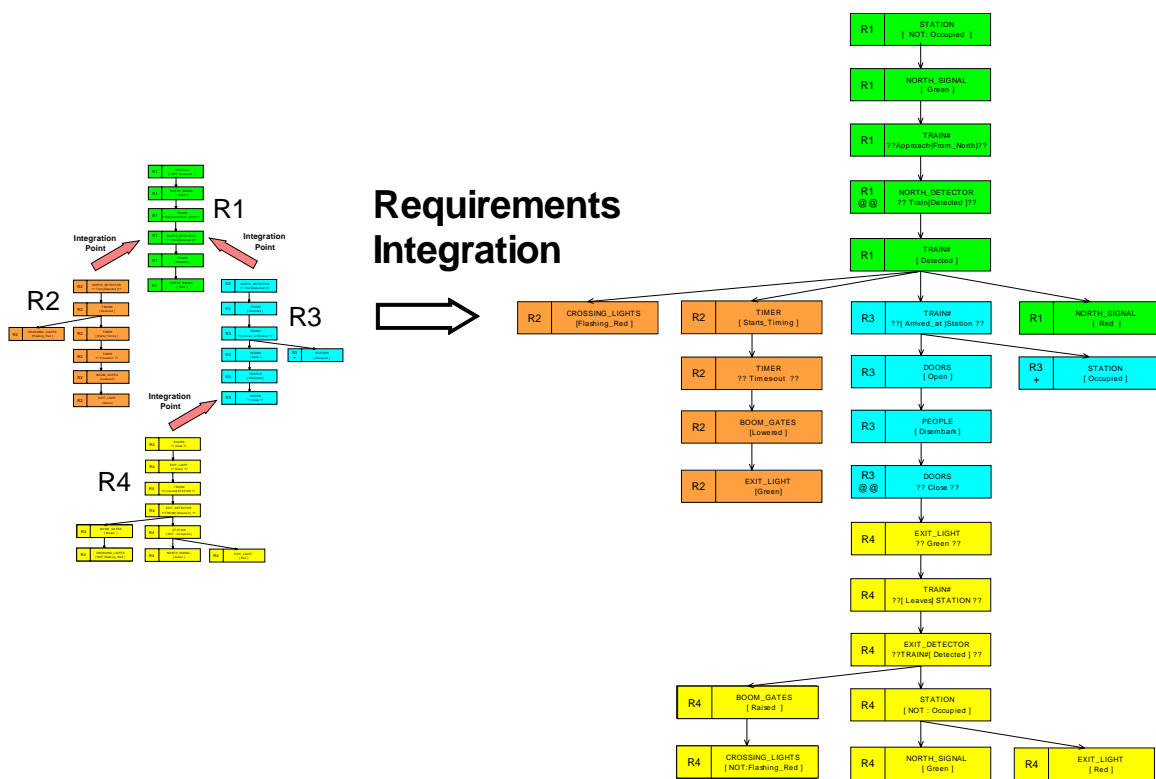


Figure 9. Integration of all requirements behavior trees for Train Station System.