

“Integrare”, a Collaborative Environment for Behavior-Oriented Design

Lian Wen¹, Robert Colvin², Kai Lin¹, John Seagrott¹, Nisansala Yatapanage¹ and Geoff Dromey¹

¹ Griffith University, Brisbane, Qld, Australia

² University of Queensland, Brisbane, Qld, Australia

{l.wen, j.seagrott, g.dromey}@griffith.edu.au, {robert, nisansala}@itee.uq.edu.au, kai.lin@student.griffith.edu.au

Abstract. In this paper, we introduce a new cooperative design and visualization environment, called “Integrare”, which supports designers and developers in building dependable, component-based systems using a new behavior-oriented design method. This method has advantages in terms of its abilities to manage complexity, find defects and make checks of dependability. The environment integrates and unifies several tools that support multiple phases of the design process, allowing them to interact and exchange information, as well as providing efficient editing capabilities. It can help formalize individual natural language functional requirements as Behavior Trees. These trees can be composed to create an integrated tree-like view of all the formalized requirements. The environment manages complexity by allowing multiple users to work independently on requirements translation and tree editing in a collaborative mode. Once a design is constructed from the requirements, it can be visually simulated with respect to an underlying operational semantics, and formally verified by way of a model checker.

Keywords: behavior-oriented design, behavior tree, software environment.

1 Introduction

Software tools, from editors and compilers to software engineering environments, which are integrated collections of different tools, have been developed and used from the very early days of software engineering [4]. As software systems are becoming larger and more complex, selecting the right tools and environments is critical to the quality and speed of developing these systems [6]. In this paper, we introduce a new collaborative environment “Integrare”, which can be used throughout multiple phases in the software design cycle, such as requirement engineering, simulation, formal specification, and model checking.

Integrare is built to support the Behavior Tree (BT) design method [1], which is a process that constructs a component-based software design from the system's functional requirements. This process is a systematic method for translating informal natural language functional requirements into a formal BT representation, in a

straightforward and traceable manner. Validation of the system model is one of the most important tasks in developing software that meets the client's needs, and Integrare supports this in a rigorous manner by including simulation and model checking facilities, in contrast to many commercially available modeling tools based on UML [24]- [27] and requirement engineering tools [28]-[30]. The first version, which has been released for internal testing, includes the following functions:

- Visio-styled user interface.
- A collaborative (multi-user) working mode.
- A Requirement Translation Assistant (RTA).
- Simulation.
- Translation of BT to SAL for model checking.

Integrare was developed using C++, employing Visual Studio [21] and Microsoft Foundation Classes [22]. It uses XD++ [23] as the library to support graphical editing. The architecture is a hybrid of model-view [33] and event-driven [32] architectures.

The paper is organized as follows: in section 2 we briefly introduce the BT design process and notations. The architecture and GUI are described in section 3, and from section 4 to section 7, we present four major features of the tool, which are its collaborative working mode, the requirement translation assistant, simulation, and SAL translation. Related work is discussed in section 8.

2 The Behavior Tree design approach

The Behavior Tree (BT) approach is a software design process that constructs a component-based software design from the system's functional requirements. This process is a systematic method for translating informal natural language functional requirements into a formal BT representation, in a straightforward and traceable way [1] [7]. The constructed BT can be used to support different stages and different aspects of software engineering such as requirements engineering [11], architecture and component design, software change [3], architecture normalization [2], model checking [9] safety [14], reliability issues [10], verification [15] and simulation.

Compared with UML, independent researchers find that the lack of precise [40], formal [36] and unambiguous [37] semantic models is one of the major difficulties in checking the consistency between different UML diagrams [38], translating UML into formal languages [39], and simulating UML models [40]. In contrast, the formal semantics of the BT notation has been stressed from the beginning; a formal semantic language Behavior Tree Specific Language (BTSL) has been developed [11], and a BT can be automatically translated into formal languages such as CSP [9] and SAL [10], and described by a metamodel [8]. Even though a BT is a formal specification, unlike formal languages such as CSP, SAL or B notation [41], the flowchart-styled graphic notation of BT can be easily understood by non-experts. Therefore, the BT notation has both advantages as a formal language with precise semantics so it can be mechanically checked, analyzed and simulated, as well as a soft and casual modeling [5] that non-technical stakeholders find appealing.

The BT approach also provides a systematic way to transform the natural language described user requirements into component-based designs, in contrast to approaches

based on UML use case diagrams, PLUSS [42], or interdependency graphs (SIG) [43]. The transformation process follows three steps [1]. Firstly, each individual functional requirement is translated into one or more corresponding Requirement Behavior Tree(s) (RBT). This process, aided by tools such as the RTA (see section 5), is focused on traceability and preserving the intention of the natural language requirements. Secondly, the RBTs are integrated into a Design Behavior Tree (DBT). The DBT may be validated by the client, both by-hand and with the aid of a visual simulation tool. The DBT may also be model-checked to formally verify that it fulfills safety or performance requirements. Finally design diagrams are projected out from the DBT. Details of the BT approach can be found from [16].

3 Architecture and the GUI

The architecture of Integrare, shown in Fig. 1, can be described from two different aspects. One is the static aspect that focuses on the composition and structure of the architecture, where the architecture is similar to a model-view architecture [24]. The other aspect is dynamic, describing the runtime working flows of the system, from which the architecture is like an event-driven architecture [32].

The model-view architecture includes two major parts: the data model and the views. The data model holds the application data and provides interfaces to query and modify the data, while the views are collections of ways to present the data stored in the data model. In the event-driven architecture, all the runtime operations should be consequences of events.

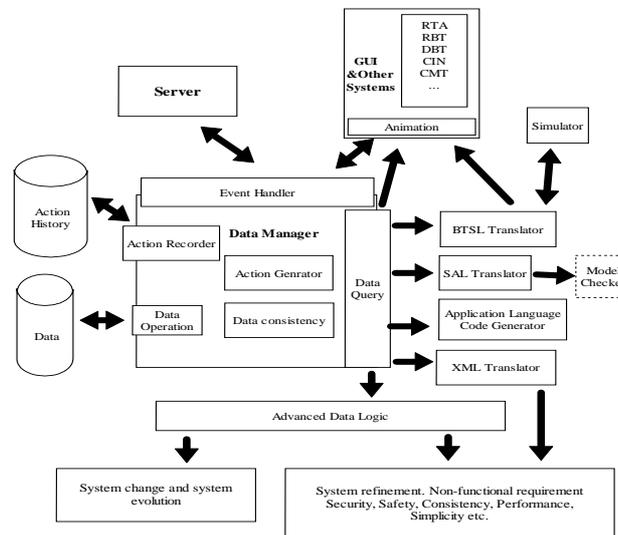


Fig. 1. The architecture of Integrare.

In Integrare, the center component, called data manager, includes 6 sub components: “event handler”, “data query”, “data operation”, “action recorder”, “action generator” and “data consistency”. The “data operation” is the only

component that can directly access the raw data; the “data query” is the public interface for other components to query the data of the system. The only way to modify the data is to post events to the “event handler”. An event can be triggered from the GUI, other systems or the server, which support cooperative design. When the “event handler” receives an event, it will pass the event to the “action generator”; with the help of the “data consistency”, the “action generator” may generate a sequence of actions; these actions will be recorded by the “action recorder” into the action history and also executed by the “data operation” to modify the data. The component “action history” records the completed sequence of the executed actions. This information can be used to reproduce the data images of a system in the different stages, and it may be helpful to study the evolution of a system.

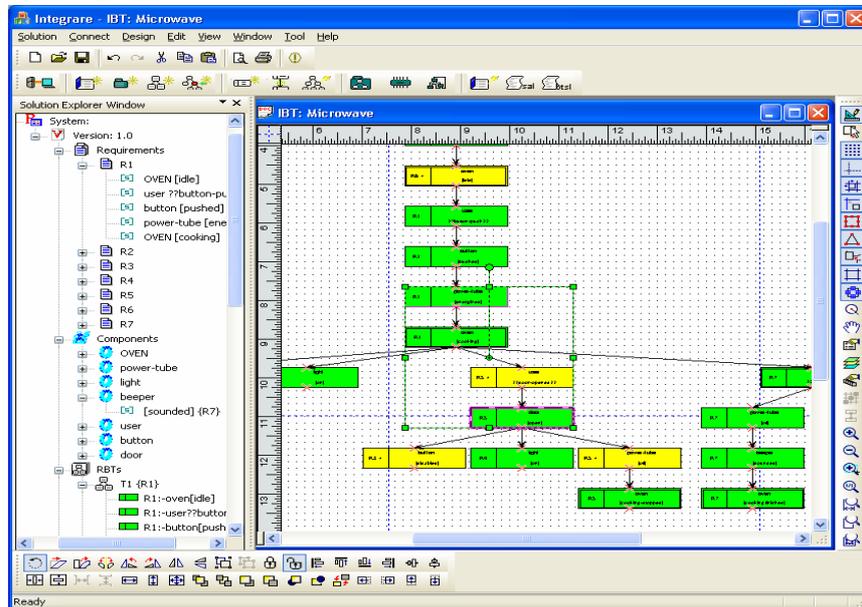


Fig. 2. The GUI of Integrare.

Visualization is an essential part of Integrare, which uses a graph editing library XD++ [23] to power the GUI. Fig. 2 is a screenshot of Integrare. It supports many standardized GUI functions such as zooming, cut/paste, layout arrange and redo/undo. People who have experience with other graph editing tools such as Visio or Smartdraw [31] will find it is easy to use Integrare.

4 Collaborative Mode

The Collaborative mode is important for designing large systems, which usually require a team of people to work simultaneously.

To meet the requirement of high responsiveness in the slow network environment, replicated architecture is adopted in Integrare. Shared documents are replicated at the

local storage of each collaborating site, so that operations can be performed at local sites immediately and then propagated to remote sites. However, concurrent editing in the replicated architecture may cause three kinds of inconsistency problems [35]: (1) causality violation: operations may arrive and be executed out of their natural cause-effect order; (2) divergence: operations may arrive and be executed at different sites in different orders; and (3) intention violation: the actual execution effect of an operation may be different from the intention of this operation. Moreover, in collaborative Integrare, many constraints must be maintained automatically.

It is obvious that if operations are executed in the same order at each collaborating site, convergence is guaranteed. In Integrare, we separate two different types of actions; the first type will not change the data model of BT approach, and the second type of actions will change the data model related to BT approach. To improve the performance, only the second type of actions will be synchronized by the server.

5 Requirement Translation Assistant (RTA)

The previous two sections have introduced the architecture and the collaborative working mode of Integrare. In this and following sections we will introduce the functionalities of BT approach supported by Integrare. The first step in developing a requirements specification using the BT approach is to translate the natural language requirements. This involves extracting all of the behavioral, structural and compositional information out of the requirements. The RTA facilitates this task.

In carrying out translation, the requirements for the system can be split up amongst multiple developers. An individual only needs to have the requirements that have been allocated to him/her. The cooperative environment provides functions ensure all team members use a common vocabulary when integration occurs.

An individual can take each requirement that they have been given and read them thoroughly, so obtaining a full understanding of the requirement. It is then necessary to move through the natural language requirement and identify components, behavior and behavior types. These individual items can be identified separately, but it is then necessary to link behavior to a component and a behavior type. This provides the information necessary to create a BT node. This process results in one or more BT nodes for each natural language requirement. Once we have a set of BT nodes for a given natural language requirement, we can export them to the main BT editing tool, where they can be joined together to form BTs.

6 BTSL and Simulation

One of the key capabilities for developing a behavioral model of a system is the ability to rapidly validate that the system being designed behaves in the intended manner. As such Integrare includes an interface to a BT simulator, BTsim. The user can observe each step the system takes (each step typically corresponds to the "execution" of one node), with the most recently executed node highlighted. The user

can also observe the values of the components after each step, and can provide safety properties for the simulator to check.

The simulator is written in the logic programming language Mercury [44], and encodes the rules of an operational semantics for BTs [11]. The simulator takes as input a BT, initial values of the components, and an optional list of safety properties to check. The simulator operates in two main modes: it can generate a random trace (sequence of nodes) of the system, either interactively or automatically; and it can exhaustively generate all traces. In addition, at each step it checks that the properties it was given on initialization still hold.

To interact with the simulator, Integrare first translates the BT into the simulator syntax. This syntax is a logic programming term, based on a recursive, tree data structure. The translation process walks down the tree from the root node recursively building up the term. Each node is augmented with its internal node label within Integrare; this provides the mechanism by which BTsim communicates to Integrare which node is executed at each simulation step. Once the translated tree has been sent to BTsim, it immediately executes an atomic step, according to the operational semantics, and returns the identifier of the executed node. On reading this output, Integrare highlights the appropriate node, and displays the new value of the components. If a property has been violated, the simulation halts, and the violated property is shown to the user. The process is repeated until the tree finishes execution, or the user stops the simulation. The user can step through the execution one step at a time, or can set a time interval for steps to be executed (and can mix the two approaches).

Simulation not only provides a way for the modeler to concretely observe dynamic behavior, but also to quickly check that the model maintains certain properties, for instance, that a component never reaches some erroneous state, or that eventually a component reaches a healthy state.

7 SAL Translation and Model Checking

Recent approaches for the verification of system designs have involved formal methods, including model checking. Model checking is a process in which a model of the system is verified against specified properties, such as safety requirements [45]. The model checker either proves that each property holds for the model, or provides a counterexample, which describes the steps which lead to the violation of the property.

Model checking usually requires expert knowledge of the input language, making it difficult to use for those without experience in formal methods. For this reason, a translator was created for automatic translation from BTs to the input language of the SAL framework [46]. SAL is a suite of tools which provide various capabilities for the analysis of concurrent systems, including symbolic and bounded model checkers.

A set of syntax rules for BTs have been devised, along with a translation scheme from BTs to the SAL language corresponding to each rule. BT nodes are translated into transitions in the SAL language, with variables representing components and messages. State-realizations are translated into updates to the variables, while BT guards and selections are translated into tests on the state of variables. Program

counters provide the flow of control and enable branching and other BT concepts, such as thread kill and reversion, to be represented.

These rules form the basis of the translator, which first parses the BT according to the syntax rules to ensure that the BT is syntactically correct. The sequence of syntax rules is then used for the translation phase. The translation rule corresponding to each syntax rule in the sequence is applied, producing the SAL model.

8 Related Work

Many commercial tools and environments currently exist for modeling in UML (eg, [26] [27]). Such tools typically focus on the presentation of the models and generating code from them. The BT method covers a larger range of the software development process, and hence Integrare, to support the method, contains features not found in UML-based tools. In particular, BTs aid in the construction of models in a systematic, traceable way from natural language [1]. This is covered in Integrare by the Requirements Translation Assistant. As part of the validation process, models can be dynamically simulated within Integrare as they are developed, giving immediate feedback on how different requirements interact. The BTs may also be model checked against safety properties of the system. These two features crucial to validation, simulation and model checking, are missing from all commercial environments we surveyed. The other distinguishing feature of Integrare is that it allows multiple users to edit the same BT in real-time.

In the research community, the SOFL method [47] is supported by a range of tools. Integrare combines a similar range of tools into one environment, allowing for quick and easy exchange of information. Integrare is also different in that it uses a single notation, BT, across all facets of software design.

Compared with existing BT environments such as BTE [17], CoGSE [18] and GSET [20], Integrare covers more aspects of the development process and aim for real applications (a few companies has shown interest in using Integrare in their large-scale software projects). The other environments are generally for research purpose and usually focus on one or a few particular phases of the design process, for example, BTE is generally for model checking, the CoGSE is used for testing the collaborative working mode, and the GSET is for modeling software change.

9 Conclusions

In this paper we have described a prototype tool that supports the BT program development framework. It incorporates several tools, starting from a RTA that begins the process of formalizing a natural language specification, through to tools for simulating and model checking designs. They have been unified under a common, easy-to-use graphical interface and, crucially, the interface supports real-time cooperative design and visualization. This increases productivity by allowing concurrent development without the need to separately merge individual work. The unified tool provides a sound base for future research and industrial applications.

Integrare was from the outset designed to progressively accommodate new functionality as it is developed. In addition to a versioning system, there are two key areas in which Integrare will be extended: support for Composition Trees (CT) and source code generation. CT works as a supporting platform, on which the BTs will be more precisely defined. Just as BTs can be automatically translated to formal languages such as SAL, it is also possible to translate them into implementation languages such as Java or C++. Some unpublished research has been done on this subject already and the results will be integrated into Integrare in the future.

Acknowledgments. The authors would like to thank Ankur Choudhary, Diana Kirk, Maria Aneiros, Saad Zafar and Lars Grunske for their contribution on this environment. This work is supported by ACCS (ARC Center for Complex Systems).

References

1. Dromey, R.G., "From Requirements to Design: Formalising the Key Steps", IEEE International Conference on Software Engineering and Formal Methods, 2003, pp. 2-11.
2. Wen, L., Dromey, R.G., "Architecture Normalization for Component-based Systems", Electronic Notes in Theoretical Computer Science, vol.160, 2006, pp. 335-348.
3. Wen, L., Dromey, R.G., "From Requirements Change to Design Change: A Formal Path", SEFM 2004, pp. 104-113
4. Harrison, W., Ossher, H., Tarr, P., "Software engineering tools and environments: a roadmap", the Conference on The Future of Software Engineering, 2000, pp. 261 – 277
5. Nuseibeh, B., Easterbrook, S., "Requirement Engineering: a Roadmap", The Future of Software Engineering , ACM Press 2000
6. Bruckhaus, T., "The impact of inserting a tool into a software process", the conference of the Centre for Advanced Studies on Collaborative research: SE - Volume 1, 1993,
7. Glass, R.L., "Practical Programmer: Is This a Revolutionary Idea, or Not?", Communications of the ACM. 47(11), 2004, pp. 23-25.
8. Gonzalez-Perez, C., Henderson-Sellers, B., Dromey, G., "A Metamodel for the Behavior Trees Modelling Technique", ICITA 05, 2005, pp. 35-39
9. Winter, K., "Formalising Behavior Trees with CSP", International Conference on integrated Formal Methods, IFM'04, 2004, pp. 148-167.
10. Grunske, L., Lindsay, P., Yatapanage, N., and Winter, K., "An Automated Failure Mode and Effect Analysis Based on High-Level Design Specification with Behavior Trees", the Fifth International Conference on Integrated Formal Methods (IFM'05), 2005, pp. 129-149.
11. Colvin, R., Hayes, I.J., "A Semantics for Behavior Trees", ACCS Technical Report, No. ACCS-TR-07-01, ARC Centre for Complex Systems, April 2007.
12. Dromey, R.G, Powell, D., "Early Requirements Defects Detection", TickIT International, 4Q05, 2005, pp. 3-13.
13. Dromey, R.G., "Scaleable Formalization of Imperfect Knowledge", 1st Asian Working Conference on Verified Software (AWCVS'06), 2006, Macau.
14. Zafar, S., Dromey, R. G., "Integrating Safety and Security Requirements into Design of an Emedded System", Asia-Pacific Software Engineering Conference, 2005, pp. 629-636.
15. Zafar, S., Winter, K., Colvin, R., Dromey, R. G., "Verification of an Integrated Role-Based Access Control Model", 1st Asian Working Conderence on Verified Software, 2006
16. Behavior Engineering, <http://www.behaviorengineering.org/index.php>, 2007
17. Smith, C., Winter, K., Hayes, I., Dromey, R.G., Lindsay, P., Carrington, D., "An Environment for Building a System Out of its Requirements", ASE, 2004, pp. 398-399.

18. Lin, K., Chen, D. Sun C. and Dromey, R.G., "Maintaining constraints in collaborative graphic systems: the CoGSE approach", 9th European Conference on CSCW, 2005.
19. Lin, K., et al.: "Maintaining multi-way dataflow constraints in collaborative systems", Int. Conference in Collaborative Computing: Networking, Applications and Worksharing, 2005
20. Wen, L., "What is GSET and what it can do", <http://www.sqi.gu.edu.au/gse/tools/gset.html>
21. Visual Studio 2005, <http://msdn.microsoft.com/vstudio/> (2007)
22. MFC <http://www.visionx.com/mfcpro/> (2007)
23. XD++, <http://www.ucancode.net/> (2007)
24. Poseidon, <http://www.gentleware.com/> (2007)
25. Altova UModel, <http://www.altova.com> (2007)
26. MagicDraw, <http://www.magicdraw.com/> (2007)
27. IBM-Rational Rose, <http://www-306.ibm.com/software/awdtools/developer/rose/>
28. Analyst Pro, <http://www.analysttool.com/> (2007)
29. Borland Caliber, <http://www.borland.com/us/products/caliber/>
30. Telelogic DOORS. <http://www.telelogic.com/products/doors/doors/index.cfm> (2007)
31. SmartDraw <http://www.smartdraw.com/>
32. Gerndt, R., Ernst, R., "An Event-Driven Multi-Threading Architecture for Embedded Systems", the 5th Int. Workshop on Hardware/Software Co-Design, 1997, pp. 29-33
33. Krasner, G. E. and Pope, S. T. 1988. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. J. Object Oriented Program. 1, 3, Aug. 1988, pp. 26-49.
34. Sun, C. and Chen, D., "Consistency maintenance in real-time collaborative graphics editing systems", ACM Transactions on Computer-Human Interaction, Vol. 9, No.1, 2002, pp.1-41.
35. Sun, C., et al.: 'Achieving convergence, causality-preservation, and intention-preservation in real-time cooperative editing systems', ACM Transactions on Computer-human Interaction, 5(1), Mar. 1998, pp. 63-108.
36. Simmonds, J., Bastarrica, M.C., "A tool for automatic UML model consistency checking", the 20th IEEE/ACM international Conference on Automated software engineering, 2005,
37. Malgouyres, H., Motet, G., "A UML model consistency verification approach based on meta-modeling formalization", ACM symposium on Applied computing, 2006
38. Vidal, J.S., Malgouyres, H., and Motet, G., "UML2.0 consistency rules identification", The International Conference on Software Engineering Research and Practice, SERP, 2005,
39. McUmber, M., E., Cheng, B.H.C., "A General Framework for Formalizing UML with Formal Language", the 23rd International Conference on Software Engineering, 2001.
40. Cavarra, A., Riccobene, E., et al.: "A Framework to Simulate UML Models: Moving from a Semi-formal to Formal Environment", ACM symposium on Applied computing, 2004
41. Bouquet, E., Legeard, B., Peureux, F. and Torrebore, E., "Mastering Test Generation from Smart Card Software Formal Models", CASSIS'04, volume 3362, 2004, pp. 70-85
42. Eriksson, M., Morast, H., Börstler, J., "The PLUSS toolkit -- extending telelogic DOORS and IBM-rational rose to support product line use case modelling", Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, 2005, pp 300-304
43. Cooper, L., Chung, L., "Managing Change in OTS-Aware Requirements Engineering Approach", The 2nd international workshop on Models and processes for the evaluation of off-the-shelf components, 2005, pp. 1-4
44. Somogyi, Z., Henderson, F.J., et al.: "Mercury, an efficient purely declarative logic programming language", the 8th Australasian Computer Science Conference, pp 499-512,
45. Clarke, E.M., Wing, J.M., Formal Methods: State of the Art and Future Directions, ACM Computing Surveys, Vol. 28, Issue 4, Dec. 1996, pp. 626-643.
46. Bensalem, S., Ganesh, V., Lakhnech, Y., Muñoz, C., Owre, et al.: "An Overview of SAL", Fifth NASA Langley Formal Methods Workshop (LFM 2000), 2000, pp.187-196.
47. Liu, S., "Formal Engineering for Industrial Software Development using the SOFL Method", Springer Verlag, 2004, ISBN 3-540-20602-7