Aristotle University of Thessaloniki

Faculty of Sciences

School of Informatics

Doctoral Thesis

Emmanouela Stachtiari

Correct-by-construction model based design of systems and software

Supervisor: Assist. Professor Panagiotis Katsaros

October 24, 2018

ΑΡΙΣΤΟΤΕΛΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΟΝΙΚΗΣ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Εμμανουέλα Στάχτιαρη

Από κατασκευής ορθή σχεδίαση συστημάτων και λογισμικού βάσει μοντέλων

Επιβλέπων: Επίκ. Καθηγητής Παναγιώτης Κατσαρός

24 Οκτωβρίου 2018

Emmanouela Stachtiari

Correct-by-construction model based design of systems and software

Doctoral Thesis

Submitted to the Department of Informatics, Faculty of Sciences, Aristotle University of Thessaloniki
Defense Date: 24/10/2018

Examination Committee

Panagiotis Katsaros, Assistant Professor, Department of Informatics, Aristotle University of Thessaloniki - *Supervisor*

Ioannis Stamelos, Professor, Department of Informatics, Aristotle University of Thessaloniki - *Advisory Committee Member*

Eleftherios Angelis, Professor, Department of Informatics, Aristotle University of Thessaloniki - *Advisory Committee Member*

Saddek Bensalem, Professor, Université Grenoble Alpes - *Examiner*

Alexandros Chatzigeorgiou, Professor, Department of Applied Informatics, University of Macedonia - *Examiner*

Andreas Symeonidis, Associate Professor, Department of Electrical Engineering, Aristotle University of Thessaloniki - *Examiner*

Simon Bliudze, Research Scientist, Inria Nord, Lille - *Examiner*

Ph.D. Thesis Title:
Correct-by-construction model based design of systems and software

“The approval of this Ph.D. Thesis by the School of Informatics of the Aristotle University of Thessaloniki does not imply acceptance of the opinions of the author” (L. 5343/1932, Article 202, par. 2)

ARISTOTLE UNIVERSITY OF THESSALONIKI

Abstract

Faculty of Sciences
School of Informatics

Doctor of Philosophy

Correct-by-construction model based design of systems and software

by Emmanouela Stachtiari

Software systems of today are complex and wide in scope in order to meet industry needs. Checking correctness in such systems through validation testing reveals flaws late, in which case a backtracking in the development phase is required to pursue a new design and implementation. Instead, the design of complex systems should rely on early validation of requirements, which could take place at the design phase to ensure that implementation will end with a correct product.

The aim of this thesis is to introduce systematic approaches that fit into a rigorous design flow for software systems. To this end, the research question is as follows: to what extent can a software system be designed towards meeting its functional specification while avoiding a-posteriori verification as much as possible (correctness-by-construction). In this context, a software system should be represented by a design model, while its functional specification is a set of formal properties which should be implied by the model's structure and behavior.

The research question is answered through a tool-supported rigorous design flow that relies on the incremental construction of models using the BIP (Behavior-Interaction-Priorities) component framework. We focus on how to derive and validate a functional application model from a set of requirements or from programs written in an application programming language. Our techniques address the following challenges:

1. Early validation of requirements and system design, to eliminate the need for a posteriori verification and reduce the validation testing during the late stages of development. The effectiveness of the process and the tool

support was evaluated on a set of requirements for the control software of the CubETH nanosatellite and an extract of software requirements for a Low Earth Orbit observation satellite. Through our approach, we managed to ensure all requirements by construction and limit the needs for system verification.

2. Automated generation of functional BIP application models that preserve the semantics of programs written in programming languages with nesting syntax. We focused on the correctness of web service compositions written in BPEL and we applied our approach and the associated tool to a benchmark of real BPEL programs with diverse complexities of model creation and verification.

3. Maintain the consistency between the BIP model and the application code across the rigorous design steps. To this end, we propose using a suitable domain-specific language. We addressed the research challenge in the context of resource-constrained REST IoT application design, for WPAN systems with nodes running the Contiki OS. The application code is rendered correct by construction, in the sense that it corresponds exactly to the validated model.

ΑΡΙΣΤΟΤΕΛΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΟΝΙΚΗΣ

Περίληψη

Σχολή Θετικών Επιστημών
Τμήμα Πληροφορικής

Διδακτορικό Δίπλωμα

Από κατασκευής ορθή σχεδίαση συστημάτων και λογισμικού βάσει μοντέλων

από την Εμμανουέλα Στάχτιαρη

Τα τελευταία χρόνια παρατηρείται αύξηση της πολυπλοκότητας και του εύρους των εφαρμογών των συστημάτων λογισμικού, ενώ οι ανάγκες της βιομηχανίας εξελίσσονται συνεχώς. Η εκπλήρωση των λειτουργικών και μη λειτουργικών απαιτήσεων καθορίζει αυτό που αντιλαμβανόμαστε ως ορθότητα της σχεδίασης του συστήματος. Οι λειτουργικές απαιτήσεις ορίζουν την αναμενόμενη συμπεριφορά του συστήματος, ενώ οι μη λειτουργικές απαιτήσεις ορίζουν περιορισμούς σε ποιοτικές πτυχές που σχετίζονται με το χρόνο εκτέλεσης και την αξιοπιστία. Η ικανοποίηση των απαιτήσεων επαληθεύεται με δοκιμές επικύρωσης (validation testing) στα τελευταία στάδια της ανάπτυξης του συστήματος. Η επαλήθευση της ορθότητας μπορεί να γίνει παρόλα αυτά και σε πρώιμα στάδια, αν η σχεδίαση αποτυπώνεται σε ένα αφαιρετικό μοντέλο της συμπεριφοράς. Καθώς όμως η ορθότητα δεν είναι απολύτως εφικτή, μπορούμε να επιδιώξουμε ιχνηλάσιμες διαδικασίες σχεδίασης, δηλαδή μία προσέγγιση αυστηρής σχεδίασης, μέσω της οποίας είναι εφικτή η τεκμηρίωση των σχεδιαστικών επιλογών ως προς τις συνέπειές τους.

Για τη διαχείριση της πολυπλοκότητας στη σχεδίαση των συστημάτων, οι αυστηρή σχεδίαση μπορεί να στηριχθεί στη σύνθεση συστατικών λογισμικού και υλικού (component-based design). Τα συστήματα συστατικών αναπτύσσονται συναρμολογώντας συστατικά που αλληλεπιδρούν μεταξύ τους. Ο ταυτοχρονισμός είναι ένα εγγενές χαρακτηριστικό των συστημάτων συστατικών. Η ταυτόχρονη πρόσβαση σε κοινόχρηστους πόρους μπορεί να προκαλέσει διενέξεις και προβλήματα, όπως

τα αδιέξοδα, που υπονομεύουν την ορθότητα της σχεδίασης. Για την επαλήθευση της ορθότητας συστημάτων συστατικών, αρκεί να ελέγχεται η συμπεριφορά κάθε συστατικού για να διαπιστωθεί αν ικανοποιεί τις αναγκαίες υποθέσεις για την εκπλήρωση των απαιτήσεων του συστήματος συνολικά. Αν οι συγκεκριμένες υποθέσεις ικανοποιούνται, τότε είναι δυνατή η ενσωμάτωση του συστατικού στο σύστημα.

Η Αυστηρή Σχεδίαση Συστημάτων [1] (ΑΣΣ) βασίζεται σε μια ιχνηλάσιμη και επαναληπτική διαδικασία με χρήση τυπικών μοντέλων (i) του λογισμικού εφαρμογών, (ii) της πλατφόρμας εκτέλεσης και (iii) του εξωτερικού περιβάλλοντος του συστήματος. Απώτερος στόχος είναι η παραγωγή μιας αξιόπιστης και βελτιστοποιημένης υλοποίησης του συστήματος από τα μοντέλα σχεδίασης μέσω μιας ακολουθίας μετασχηματισμών, που διατηρούν τη σημασιολογία εκτέλεσης.

Η Αυστηρή Σχεδίαση Συστημάτων αποτελείται από τέσσερα βήματα:

Βήμα 1 Το *λειτουργικό μοντέλο της εφαρμογής* ορίζεται βάσει μίας προδιαγραφής της λειτουργικής συμπεριφοράς των συστατικών και των περιορισμών συντονισμού, που επιβάλλονται από τις αλληλεπιδράσεις τους. Σε αυτό το στάδιο ελέγχονται οι ιδιότητες συμπεριφοράς, όπως η απουσία αδιεξόδου, μέσω ανάλυσης των αλληλεπιδράσεων των συστατικών.

Βήμα 2 Ένα *αφαιρετικό μοντέλο του συστήματος* κατασκευάζεται συνδυάζοντας το μοντέλο εφαρμογής του προηγούμενου βήματος με ένα μοντέλο της πλατφόρμας εκτέλεσης και την αντιστοίχιση των συστατικών της εφαρμογής σε συστατικά της πλατφόρμας. Σε αυτό το στάδιο, μπορεί να αξιολογηθεί η απόδοση του συστήματος μέσω προσομοίωσης που λαμβάνει υπόψιν τα χαρακτηριστικά της πλατφόρμας εκτέλεσης.

Βήμα 3 Ένα *λεπτομερές μοντέλο του συστήματος* επιτυγχάνεται με μετασχηματισμό του αφαιρετικού μοντέλου, ενσωματώνοντας πρωτογενή στοιχεία επικοινωνίας της πλατφόρμας εκτέλεσης για την αλληλεπίδραση συστατικών. Σε αυτό το στάδιο, οι μηχανισμοί αλληλεπίδρασης υψηλού επιπέδου εκλεπτύνονται και αντικαθίστανται από πρωτογενή στοιχεία επικοινωνίας, όπως αποστολή μηνυμάτων ή χρήση κοινής περιοχής μνήμης.

Βήμα 4 Στο τέλος παράγεται εκτελέσιμος κώδικας για κάθε συστατικό επεξεργασίας της πλατφόρμας στόχο.

Το BIP (Behavior, Interaction, Priority) [2] είναι ένα πλαίσιο συστατικών με κατάλληλη υποστήριξη τη διαδικασία ΑΣΣ. Επιτρέπει το διαχωρισμό σχεδιαστικών πτυχών σε δύο επίπεδα: καταρχήν, σε ένα μοντέλο BIP η συμπεριφορά προδιαγράφεται ξεχωριστά από τις αλληλεπιδράσεις των συστατικών και τις προτεραιότητές τους. Επίσης, διατίθενται γεννήτριες BIP κώδικα με αφετηρία διάφορες γλώσσες προγραμματισμού, γεγονός που καθιστά το BIP ένα πλαίσιο σημασιολογίας που ενοποιεί τη σημασιολογία των διαφόρων γλωσσών, ιδιαίτερα αποτελεσματικό για τη σχεδίαση συστημάτων με λογισμικό και υλικό. Επιπλέον, οι απλουστευμένες δομές συντονισμού των συστατικών BIP καθιστούν τη διαδικασία σχεδίασης ιχνηλάσιμη ως προς τις επιπτώσεις των σχεδιαστικών επιλογών. Με τα λίγα πρωτογενή στοιχεία σύνταξης της γλώσσας η περιγραφή του συντονισμού των συστατικών γίνεται με ένα φυσικό και άμεσο τρόπο.

Στη γλώσσα BIP είναι εφικτή η υλοποιήση τεχνικών από κατασκευής ορθότητας, μέσω της διατήρησης των ιδιοτήτων ορθότητας του μοντέλου κατά τους μετασχηματισμούς που αυτό υφίσταται στα διάφορα βήματα της ΑΣΣ. Οι μετασχηματισμοί μοντέλου θεμελιώνονται με σχέσεις εκλέπτυνσης (refinement), που έχουν αποδειχθεί από κατασκευής ορθές, δηλ. διατηρούν την εξωτερικά παρατηρήσιμη ισοδυναμία και επομένως όλες τις βασικές ιδιότητες ασφάλειας. Πρακτικά, η επαλήθευση των ιδιοτήτων ασφάλειας γίνεται μόνο στα μοντέλα υψηλού επιπέδου αφαίρεσης. Για την παράκαμψη της εγγενούς πολυπλοκότητας στην επαλήθευση διατίθενται εργαλεία, που υποστηρίζουν συνθετικές τεχνικές ανάλυσης, δηλαδή μπορούμε να συμπεράνουμε την ισχύ ιδιοτήτων σε σύνθετα συστατικά BIP με βάση τις ιδιότητες που ισχύουν στα συστατικά αυτά περιέχουν.

Η διατριβή αυτή εισάγει τεχνικές από κατασκευής ορθότητας για την αυστηρή σχεδίαση συστημάτων. Ειδικότερα, εστιάζουμε στο πώς μπορούμε να παράξουμε και να επικυρώσουμε ένα λειτουργικό μοντέλο της εφαρμογής από ένα σύνολο απαιτήσεων ή από προγράμματα γραμμένα σε μια γλώσσα προγραμματισμού εφαρμογών. Οι τεχνικές μας αντιμετωπίζουν τις παρακάτω προκλήσεις:

1. Την πρώιμη επικύρωση των απαιτήσεων και της σχεδίασης του συστήματος, ώστε να εξαλειφθεί η ανάγκη μιας εκ των υστέρων επαλήθευσης και να περιοριστούν οι έλεγχοι επικύρωσης κατά τα τελευταία στάδια ανάπτυξης.
2. Την αυτόματη δημιουργία λειτουργικών μοντέλων εφαρμογών από προγράμματα γλωσσών προγραμματισμού με εμφωλευμένη σύνταξη (εστιάσαμε στη γλώσσα BPEL), διατηρώντας τη σημασιολογία των προγραμμάτων.
3. Τη διατήρηση της συνέπειας μεταξύ του μοντέλου του συστήματος και του κώδικα της εφαρμογής σε όλα τα βήματα της ΑΣΣ. Για το σκοπό αυτό, προτείνουμε τη χρήση μιας κατάλληλης γλώσσας ειδικού σκοπού (επικεντρωθήκαμε στη σχεδίαση συστημάτων περιορισμένων πόρων του διαδικτύου των αντικειμένων).

Σε ότι αφορά το (1), προτείνουμε ένα σύνολο προσαρμόσιμων προτύπων για απαιτήσεις σε φυσική γλώσσα, από τα οποία παράγονται τυπικά ορισμένες ιδιότητες, που είτε μπορεί να επιβάλλονται, είτε να επαληθεύονται μέσω επιθεώρησης ή με έλεγχο μοντέλου (model checking). Αυτή η προσέγγιση σχεδίασης βασίζεται σε ένα σύνολο προϋπάρχοντων συστατικών, που παρέχουν λειτουργίες του συστήματος. Στη συνέχεια, οι τυπικές ιδιότητας επιβάλλονται με μία σειρά μετασχηματισμών του μοντέλου της εφαρμογής, οι οποίοι εφαρμόζουν τυπικά ορισμένα πρότυπα σχεδίασης, τις λεγόμενες αρχιτεκτονικές, που αναπαριστώνται με μοντέλα BIP. Η επαλήθευση μέσω επιθεώρησης είναι δυνατή όταν οι ιδιότητες μπορούν να ελεγχθούν τοπικά στις καταστάσεις ενός μόνο συστατικού, ενώ η επαλήθευση με έλεγχο μοντέλου απαιτείται μόνο για ιδιότητες, που δεν μπορούν να εξασφαλιστούν με τα προαναφερθέντα μέσα.

Σε σχέση με το (2), εισάγουμε μια προσέγγιση για τον ορισμό συνθέσιμης σημασιολογίας, που πιστεύουμε ότι είναι εφικτή για ένα ευρύ φάσμα γλωσσών προγραμματισμού με εμφωλευμένη σύνταξη. Μια τέτοια σημασιολογία εκτέλεσης αντιμετωπίζει την πολυπλοκότητα της μετάφρασης κώδικα σε μοντέλα εφαρμογών, καθώς οι κανόνες μετάφρασης περιορίζονται στον αριθμό των πρωτογενών στοιχείων της γλώσσας και οι χρόνοι μετάφρασης κλιμακώνονται γραμμικά με το μέγεθος του προγράμματος. Η εγκυρότητα της σημασιολογίας στηρίζεται σε ιδιότητες που επιβάλλονται κατά την κατασκευή του μοντέλου. Εφαρμόσαμε επιτυχώς αυτήν την προσέγγιση για τη γλώσσα σύνθετων υπηρεσιών BPEL. Αναπτύξαμε ένα εργαλείο για τη μετάφραση προγραμμάτων BPEL σε BIP μοντέλα εφαρμογών. Το εργαλείο δοκιμάστηκε σε ένα σύνολο πραγματικών προγραμμάτων. Τα παραγόμενα BIP μοντέλα χρησιμοποιήθηκαν για να επαληθευθούν βασικές ιδιότητες ορθότητας, όπως η εξασφάλιση τερματισμού και η έγκαιρη απόκριση των υπηρεσιων, καθώς και άλλες ιδιότητες που αφορούν την εκάστοτε εφαρμογή.

Αναφορικά με το (3), αντιμετωπίσαμε την ερευνητική πρόκληση στο πλαίσιο της σχεδίασης REST (Representational state transfer) [3] εφαρμογών για συστήματα ασύρματων προσωπικών δικτύων περιοχής (WPAN) με κόμβους που χρησιμοποιούν το Contiki OS [4]. Σχεδιάστηκε μία γλώσσα ειδικού σκοπού (DSL), που εξυπηρετεί δύο σκοπούς: απλοποιεί τον προγραμματισμό εφαρμογών και διατηρεί τη συνεκτικότητα μεταξύ του κώδικα εφαρμογής και του BIP μοντέλου σε όλα τα στάδια της σχεδίασης. Η DSL παρέχει μια προγραμματιστική αφαίρεση με πρωτογενή στοιχεία σύνταξης για τον ορισμό της ροής ελέγχου και των κοινών ενεργειών πελάτη και διακομιστή, όπως η αλληλεπίδραση διεργασιών, η επικοινωνία μέσω του δικτύου και η διαχείριση πόρων. Χρησιμοποιήσαμε τη γλώσσα για να σχεδιάσουμε ένα σύστημα έξυπνου κτιρίου, του οποίου το μοντέλο επαληθεύτηκε ως προς ένα ένα σύνολο λειτουργικών και μη λειτουργικών απαιτήσεων. Έτσι, ο κώδικας εφαρμογής είναι από-κατασκευής ορθός, υπό την έννοια ότι αντιστοιχεί ακριβώς στο επικυρωμένο μοντέλο.

Publications

Journal publications

1. Stachtiari, E., Katsaros, P. *Compositional execution semantics for business process verification.* Journal of Systems and Software, Vol. 137, 217-238, Elsevier, 2017 (online: `http://doi.org/10.1016/j.jss.2017.11.003`)

2. Stachtiari, E., Mavridou, A., Katsaros, P., Bliudze, S., Sifakis, J. *Early Validation of System Requirements and Design Through Correctness-by-Construction.* Journal of Systems and Software, Vol. 145, 52-78, Elsevier, 2018 (online: `http://doi.org/10.1016/j.jss.2018.07.053`)

3. Lekidis, A., Stachtiari, E., Katsaros, P., Bozga, M., Georgiadis, C. K. *Model-based Design of IoT Systems with the BIP Component Framework.* Journal of Software: Practice and Experience, Vol. 48 (6), 1167-1194, John Wiley & Sons, 2018 (online: `http://doi.org/10.1002/spe.2568`)

Articles in conference proceedings

1. Stachtiari, E., Mentis, A., Katsaros, P. *Rigorous Analysis of Service Composability by Embedding WS-BPEL into the BIP Component Framework.* ICWS: IEEE Computer Society, 2012.

2. Stachtiari E., Vesyropoulos, N., Kourouleas, G., Georgiadis, C., Katsaros, P. *Correct-by-Constraction Web Service Architecture.* SOSE: IEEE Computer Society, 2014.

3. Lekidis, A., Stachtiari, E., Katsaros, P., Bozga, M., Georgiadis, C.-K. *Using BIP to reinforce correctness of resource-constrained IoT applications.* SIES: IEEE Computer Society, 2015.

4. Mavridou, A., Stachtiari, E., Bliudze, S., Ivanov, A., Katsaros, P., Sifakis J. *Architecture-based Design: A Satellite On-Board Software Case Study.* FACS: Springer, 2016.

Acknowledgements

At first, I would like to thank cordially the supervisor of my PhD thesis, Assistant Professor Panagiotis Katsaros, for his full trust, his guidance and the many useful tips throughout the writing of my publications and the dissertation. He has given me important lessons in scientific, career and other areas. More importantly, he worked as a role model on how to be a good scientist and person.

Furthermore, I am grateful to the members of the advisory board of my PhD thesis, namely Professor Ioannis Stamelos and Professor Eleftherios Angelis for the excellent cooperation we have had from the beginning and until now. I cannot be less thankful to all the members of my examination committee, which gave me suggestions and comments to improve my dissertation and consider new prospects as future work.

Separately, I would like to express my sincere appreciation to Simon Bliudze and Saddek Bensalem, with whom I had the privilege and pleasure to collaborate and from whom I received invaluable suggestions and ideas in functional and technical matters. I owe special thanks to Professor Joseph Sifakis for introducing me to new possibilities and a research community that has supported me throughout these years.

I am also grateful to the people with whom I had the pleasure to work during research projects. Specifically, I am indebted to Anakreon Mentis, Anastasia Mavridou and Stylianos Basagiannis for all their help and support. I cannot forget, people that I met as colleagues and other PhD candidates of our research lab, such as Georgios Chatzieleftheriou, Konstantinos Mokos and Alexios Lekidis.

Finally, all this journey would have been impossible without the most important people in my life, which are my family and friends. Therefore, I want to thank my parents, Giannis and Georgia, and my brother, Panos, for all their love and guidance that has been with me forever. Last but not least, I wish to thank my loving husband, Lefteris, who provides endless inspiration and support no matter what.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The problem of system design

In recent years there has been a tremendous increase in the complexity and scope of software systems, while industry needs evolve constantly. For instance, the expansion of the World Wide Web, and the spread of IoT have introduced the need for open, distributed and/or embedded systems, that are characterized by interactive and changing behavior, concurrency and unpredictability with respect of their run-time environment. Such complexity can be handled efficiently through modern approaches in system design, such as the integration of modular, reusable and loosely coupled components that interact to form a complete system. In addition, correctness of complex systems has to be established based on rigorous system design methodologies.

Correctness is achieved when the functional and extra-functional requirements are fulfilled within the anticipated environment of the system. Functional requirements specify what the system is expected to perform (i.e., its behavior), whereas extra-functional requirements constrain the system's performance in quantitative aspects such as time, accuracy, reliability. Requirements' satisfaction can be either tested for the final system implementation or verified against an abstract model. In a typical industrial scenario, the cost of providing correctness assurance via means of testing, debugging and verification may range from 50 to 75 percent of the system's development cost [5]. Therefore, as absolute correctness is not feasible, we aim to accountable design processes that are able to effectively handle these activities, which are so challenging and labor-intensive.

Rigorous system design methodologies rely heavily on component-based engineering for mastering design complexity. According to this paradigm, component based systems can be developed through the assembly of interacting components, which are reusable and self-contained functional modules with well-defined *provided* and *required* interfaces. Since interactions among components are not tied to their implementation details, components can be developed, versioned and deployed independently from each other, thus simplifying the system's specification, development and maintenance. In order for components to be independent, they need to be separated from the environment and other components [6].

Concurrency is an inherent characteristic of component-based systems, owing to the fact that multiple components are meant to run simultaneously by sharing the resources of the execution platform (e.g. application server, OS). For instance, embedded systems in household IoT comprise components for control decisions and components that manage the operation of sensor and actuator devices. Components in such systems interact through shared buses, memories and buffers. Simultaneous access to shared resources may cause conflicts, referred to as *resource contention*, which can lead to common issues such as deadlocks, and livelocks. Concurrency issues affect the correctness of component operations and are also observed in workflow management systems and web services.

Due to concurrency, the complexity of component-based systems increases exponentially with the number of components. Components can interact with each other in a potentially indeterminate way, leading to an overwhelming number of possible event interleavings. This kind of complexity makes it hard to establish correctness in the final system implementation. Instead, in such systems the behavior of existing components is checked to determine if they can be used to meet system requirements. First, an iterative process of specifiying and refining system requirements is necessary, so that requirements can be allocated to the components. Then, appropriate component integration should be decided to provide the necessary functionality. Afterwards, the selected components are assembled using locally-developed code [7]. In contrast to monolithic systems, where system integration is the final step of the design, in component-based systems deciding which components shall be used and how they will be integrated is a key design step.

Rigorous design methodologies are based on the utilization of software design models. In such approaches, models are treated as primary design artifacts, which are amenable to formal verification, simulation and code generation. Thus, a system's design lifecycle that uses models at early phases, i.e. after requirements specification, can provide early evidence of design correctness. Moreover, software is usually specified at multiple levels of abstraction, wherein the involved data and functions are represented at varying degrees of detail. Models can help to keep coherence and cohesion among the different design abstraction levels, by iteratively refining the model of the previous level. The ultimate goal of any such model-driven process is to finally generate, rather than implement, the necessary code.

In the context of system design, formal verification is the most common means of establishing design correctness. Figure 1.1 shows a typical flow, in which an abstract finite-state model of the system and a set of formal properties are fed to a model checking tool. The tool is able to verify the properties against the model by exploring all the reachable states of the model. If a property is found to be violated, a counterexample in the form of an execution trace is returned to the user, to help finding what causes the violation.

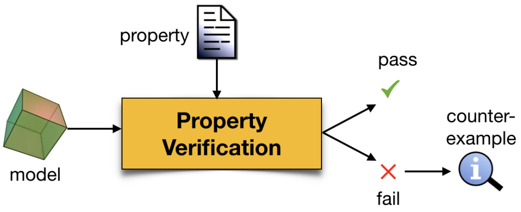


Figure 1.1: Software verification flow

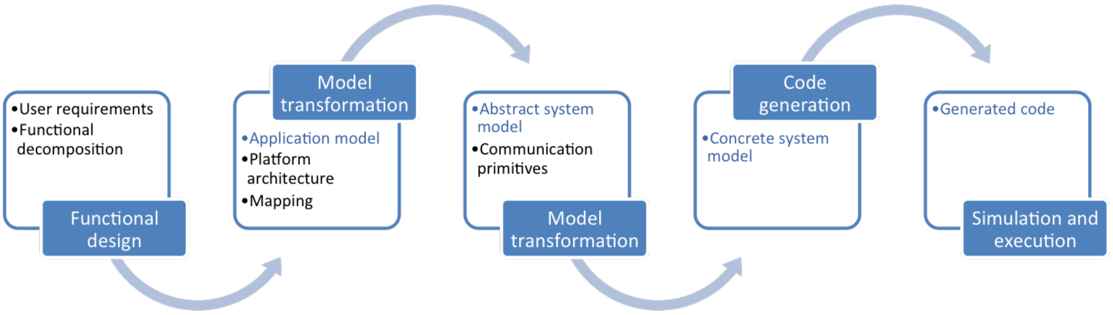


Figure 1.2: A simplified instantiation of the Rigorous System Design flow

However, the scalability limitations that model checking algorithms face in real-world systems make them impractical in many cases. Formal verification through the exploration of all possible execution paths is in fact only feasible for small- to mid-scale systems. Another problem is that counterexamples that are provided by many model checking tools are hardly traceable to a model mapped to the system's design, due to the fact that the analyzed model does not represent the actual system structure. This occurs when the modeling language is not sufficiently expressive to describe system structure. To overcome all these deficiencies, correctness by construction approaches provide means to ensure correctness based on theories and rigorous techniques for system design.

1.2 Motivation and Contributions

1.2.1 Rigorous System Design

Rigorous System Design (RSD) [1] is a design approach that is based on a formal, accountable and iterative process for deriving trustworthy and optimised implementations from models of application software, its execution platform and its external environment. The goal is to validate the design and derive a system implementation from the high-level models by applying a sequence of semantics-preserving transformations.

Figure 1.2 shows a simplified instantiation of the RSD approach with four steps:

Step 1 A *functional application model* is obtained from specifications of the functional behavior of the system components and coordination constraints imposed by their interaction. At this stage, behavioral properties, such as deadlock-freedom, can be established by analyzing component interactions.

Step 2 An *abstract system model* is obtained by combining the application model of the previous step with a model of the execution platform architecture and a mapping between application and platform components. At this stage, performance can be evaluated by simulation based on the characteristics of the execution platform components.

Step 3 A *concrete system model* is obtained by model transformation of the abstract system model, incorporating platform-specific communication primitives. At this stage, high-level interaction mechanisms are replaced by appropriate primitives provided by the execution platform, e.g. message passing or shared memory regions.

Step 4 Finally, *executable code* is separately generated for each processing element of the target platform.

The RSD flow aims at a separation of concerns, tractability of design decisions and correctness-by-construction. First, each concern is specified separately in a dedicated model using a suitable language or formalism. Second, design decisions are only taken when strictly necessary, ensuring that all design space restrictions are justified and traceable to clearly stated requirements, expressed in languages or formalisms with clear semantics. Third, properties established at any step of the design flow are preserved throughout the subsequent steps including the executable implementation. Thus, resulting systems are correct by construction.

BIP (Behavior, Interaction, Priority) [2] is a suitable component framework for supporting the RSD design flow. It enables separation of concerns at two levels: first, in a BIP model, behavior can be specified separately from interactions and priorities. Second, BIP code generators from various programming models make BIP a suitable semantic framework that unifies models and programs written in different languages, which is especially useful for mixed software/hardware systems.

Moreover, the simple coordination structure in BIP makes the whole design flow tractable with respect to design decisions. Specifically, design decisions refer to model refinements which can be represented as constraints enforced by coordination between components. Such examples are the choice of communication or scheduling protocols. Having a few powerful primitives for specifying coordination enables modeling design solutions in a natural and direct manner. A modeling language with poor expressiveness would lead to complex coordination mechanisms for establishing refinement relations and intractable models.

In BIP, correctness-by-construction can be supported by property preservation during source-to-source transformations between the different design steps. These transformations are based on refinement relations that are proven correct-by-construction, i.e., they preserve observational equivalence and therefore essential safety properties. Thus, verification of safety properties has to be applied only to high level models. To avoid inherent complexity limitations, the D-Finder [8] tool applies verification of BIP models using compositional techniques, i.e., by inferring global properties of composite components from the properties of composed components.

1.2.2 Functional modeling

The BIP approach for functional modeling takes place at the first step of the RSD flow. It consists of generating a BIP model for the algorithmic or functional aspects of the system, which may have been specified partially (e.g. as a set of requirements) or programmed in some high-level application programming language). The BIP model comprises low coupled model components, each having only one responsibility, which is performed with explicitly defined side effects on other components.

A general method exists for generating BIP models *from a given application programming language L* with well defined operational semantics, which involves the following steps [9]:

1. *Translation of atomic constructs of the source language (application components) into BIP components.* The translation requires the definition of adequate interfaces for each component and the encapsulation and reuse of their data structures and functions.
2. *Translation of coordination mechanisms between application components* into BIP connectors and priorities.
3. *Generation of a BIP component that models L's operational semantics.* This component plays the role of an engine coordinating the execution of the application components.

For generating BIP models *from a given set of requirements U* a slightly different approach is followed. First, the atomic components have to be identified based on a functional decomposition of the problem and the principle of separation of concerns. An adequate set of component interfaces can be derived from U. Then, a set of reusable coordination mechanisms can be derived, which suffices to impose the coordination specified in U. Such coordination mechanisms are called *BIP architectures* [10] and they are parameterizable, so that they can be applied to different sets of BIP components.

1.2.3 Functional correctness and thesis contributions

A system under design that meets its functional requirements is said that provides correct functionality. According to the Capability Maturity Model Integration (CMMI) project [11], a requirement specification is termed as follows:

Definition 1.2.1. (Requirement) A requirement may be:

1. a condition or capability to solve a problem or achieve an objective;
2. a condition or capability that must be met or processed by a system or a system component to satisfy a contract, standard, specification, or other formally imposed documents;

The quality of requirements can highly impact the system design and implementation, especially in safety-critical applications. Therefore, elicitation, analysis, and validation of requirements should be supported by formal means. Such methodologies should focus both on the syntactic and the semantic aspects and should also capture the dependencies among the different requirements [12]. Typically, requirements should be translated into a formal language that enables analysis and validation. The used formal language should be able to represent natural language requirements and all relevant concepts of the problem domain. Moreover, certain quality characteristics of the requirements (e.g. their consistency, realizability, completeness) have to be assessed through dedicated formal checks.

In model-based design, the declarative requirements are transformed into a model prescribing how the anticipated functionality can be realized (procedure). This problem is called proceduralization [9] and can be considered as a synthesis problem. Unfortunately, model synthesis from logical specifications has an intrinsically high complexity. Therefore, design is today mainly an empirical task relying on the expertise of the engineer, as well as on a set of principles and design solutions that have been proven effective.

Our contributions

This thesis introduces correctness-by-construction techniques suitable for rigorous system design. We specifically focus on how to derive and validate a functional application model from a set of requirements or from programs written in an application programming language. Our techniques address the following challenges in rigorous system design:

1. Early validation of requirements and system design, to eliminate the need for a posteriori verification and reduce the validation testing during the late stages of development.

2. Automated generation of functional application models that preserve the semantics of programs written in programming languages with nesting syntax (we focused on BPEL).

3. Maintain the consistency between the system model and the application code across the rigorous design steps. To this end, we propose using a suitable domain-specific language (we focused on resource-constrained IoT system design).

With respect to (1), we propose a set of customizable templates for natural language requirements, whose derived formal properties can be either enforced or verified through inspection/model checking. The design approach is based on a set of initial components that are assumed to provide the system functions. Then, property enforcement takes place through model transformations for applying known design patterns encoded in BIP models called architectures. Verification by inspection is possible when the properties can be verified locally in the states of a single component, whereas verification by model checking is needed only for properties that cannot be ensured with the aforementioned means.

With respect to (2), we introduce a compositional semantics definition approach that we believe is feasible for a wide range of programming languages with nesting syntax. Such an execution semantics minimizes the complexity of translating application code into application models, since the translation rules are limited to the number of language primitives and the translation times scale linearly to the size of the program. The semantics validity is founded on safety properties that are enforced by construction during the model synthesis. We successfully implemented this approach for the BPEL web service orchestration language. A tool for automatic translation of BPEL programs into BIP application models was developed and tested on a set of real programs. The derived application models are used to verify essential correctness properties, such as termination and service responsiveness, as well as application-specific properties.

With respect to (3), we addressed the research challenge in the context of resource-constrained REST (Representational state transfer) IoT application design [3], for WPAN (Wireless Personal Area Network) systems with nodes running the Contiki OS [4]. The developed domain-specific language language (DSL) serves two purposes: it simplifies the application programming and it maintains the coherency between the application code and the BIP model, throughout the rigorous design steps. The DSL provides a sufficient programming abstraction that consists of primitives for defining essential control flow and common client and server-side actions, such as process interaction, network communication and resource manipulation. We used the language for the specification of a smart building system, where the model was verified against a set of functional and extra-functional requirements. The application code is thus rendered correct by construction, in the sense that it corresponds exactly to the validated model.

1.3 Thesis structure

In Chapter 2, a necessary background is given regarding the BIP framework and the architecture-based design in BIP. Chapter 3 presents our proposed instantiation of the RSD process for designing correct-by-construction systems from requirements. Chapter 4 explains our methodology for defining and modeling compositional semantics for programming languages with nested syntax, using BPEL as an example. Chapter 5 shows the definition and use of a DSL language for specifying REST IoT applications that run on top of the Contiki operational system.

Chapter 2

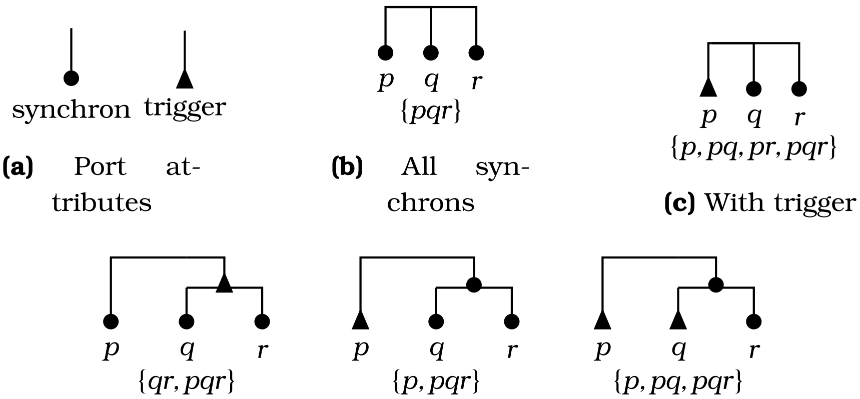
Background

2.1 The BIP framework

BIP [2] is a formal framework for building complex models by coordinating the behavior of a set of atomic model components. Behavior is defined as a transition system, extended with data and functions in C/C++. The description of coordination between components is layered. The first layer describes the interactions between components. The second layer describes dynamic priorities between interactions. BIP has a clean operational semantics that describes the behavior of a composite component as the composition of the behaviors of its atomic ones [13]. This allows a direct relation between the underlying semantic model (transition systems) and its implementation.

The atomic components are finite-state automata having transitions labeled with ports and extended with data stored in local variables. Ports form the interface of a component and are used to define interactions with other components. States denote control locations at which the components await for interaction. A transition is an execution step from a control location to another. It might be associated with a boolean condition (guard) and a computation defined on local variables. The model's global state at each execution step is given as the current control locations and the values of local variables of all atomic components.

Connectors relate ports from different subcomponents by assigning to them a synchronization attribute, which may be either trigger (represented by a triangle, Figure 2.1a) or synchron (represented by a bullet, Figure 2.1a). A connector



(a) Port attributes

(b) All synchrons

(c) With trigger

(d) Hierarchical connectors

Figure 2.1: BIP connectors and their associated interaction sets

defines a set of interactions, i.e., a non-empty set of ports. The set of interactions of each connector is based on the synchronization attributes it assigns. Given a connector involving a set of ports $\{p_1, ..., p_n\}$, the set of its interactions is defined as follows: an interaction is any non-empty subset of $\{p_1, ..., p_n\}$ which contains some port that is assigned to a trigger (Figure 2.1c); otherwise, (if all ports are assigned to synchrons) the only possible interaction is the maximal one that is, $\{p_1, ..., p_n\}$ (Figure 2.1b). The same principle is recursively extended to hierarchical connectors, where one interaction from each subconnector is used to form an allowed interaction according to the synchron/trigger typing of the connector nodes (Figure 2.1d). For instance in the third hierarchical connector shown in Figure 2.1d, port p is assigned to a trigger, whereas the binary subconnector $q - r$ is assigned to a synchron. Thus this hierarchical connector allows the singleton interaction p and any interaction that combines p with some interaction of the binary subconnector. Since the latter defines interactions q and qr, the resulting set of interactions is p, pq, and pqr.

The meaning of a BIP interaction is synchronization of ports. Recall that transitions are labelled with ports. Thus an interaction $p..q$ defines synchronization constraints on the execution of the corresponding transitions that are labelled with ports $p..q$. A BIP interaction is enabled for execution if all the corresponding transitions are enabled for execution, i.e., the current control locations of components include these transitions as outgoing transitions and all corresponding transition guards evaluate to true. The operational semantics of BIP is as follows. During the execution of a BIP interaction, all components that participate in the interaction, i.e., have an associated port that is part of the interaction, must execute their corresponding transitions simultaneously. All components that do not participate in the interaction, do not execute any transition and thus remain in the same control location.

2.2 Architecture-based design in BIP

An *architecture* in BIP is a model that characterizes the structure of the interactions between a set of component types. Such an architecture is defined with respect to a set of *parameter* components and a set of *coordinators*. The structure is specified as a relation, i.e. connectors between component ports. The components to which an architecture is applied are the *operands* that replace the architecture's parameters.

Figure 2.2 shows a BIP model for mutual exclusion between two tasks. Each component on the two outer sides models a task, which enters its critical section (i.e., the control location `work`) only when its corresponding port b_i $(i = 1,2)$ is invoked and leave it when port f_i $(i = 1,2)$ is invoked. The model has also one coordinator component C that allows the execution of b_i ports only when itself is in the `free` control location. The coordinator is in `free` after a task has left its critical section. Four binary connectors are used for the aforementioned coordination. Two connectors synchronize each of b_1, b_2 ports with the t port and two others synchronize each of the f_1, f_2 ports with the r port. The connectors essentially constrain the behavior of the system so that whenever the shared resource, managed by the coordinator, is taken by e.g., the first task, it cannot be accessed by the second task unless it is first released by the first task. Initial control locations of the components are indicated with an arc and show that both tasks are outside their critical section. Figure 2.3 shows an architecture that enforces the mutual exclusion property on two parameter components with interfaces $\{b_1,f_1\}$ and $\{b_2,f_2\}$.

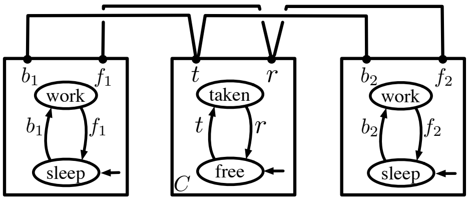


Figure 2.2: Mutual exclusion model in BIP

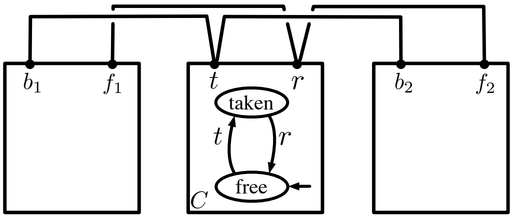


Figure 2.3: Mutual exclusion architecture

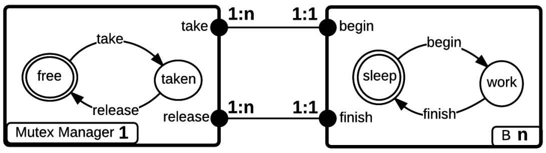


Figure 2.4: Mutual exclusion style

Composition of architectures is the conjunction of the induced synchronisation constraints. It takes the form of an associative, commutative and idempotent architecture composition operator '$\oplus$' [14], as illustrated by an example in [15]. If two architectures $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively enforce the safety properties Φ_1 and Φ_2, the composed architecture $\mathcal{A}_1 \oplus \mathcal{A}_2$ enforces the property $\Phi_1 \wedge \Phi_2$, that is, both *properties are preserved* by architecture composition. Combined application of architectures can generate deadlocks and the resulting model has to be checked for deadlock-freedom.

Although the architecture in Figure 2.3 can be applied to precisely two components, it is clear that an architecture of the same *style*—with n parameter components and $2n$ connectors—could be applied to n operand components satisfying the interface assumptions. We specify such *architecture styles* with *architecture diagrams* [16]. An architecture diagram consists of a set of *component types* with cardinality constraints for the expected number of instances and a set of *connector motifs*. Connector motifs are non-empty sets of *port types*. Each port type has a *cardinality* constraint representing the expected number of port instances per component and two additional constraints: *multiplicity* and *degree*, represented as a pair $m : d$. Additionally, each port type is typed as either trigger or synchron.

Figure 2.4 shows the architecture style of the architecture in Figure 2.3. The unique—due to the cardinality being 1—coordinator component, `Mutex manager`, manages the shared resource, while n parameter components of type

B can access it. The connector motifs have multiplicities of 1 (i.e., in 1:_) in all port types, denoting that all connectors are binary. The degrees of 1 (i.e., in :1) require that each port instance of a component of type B is attached to a single connector with the coordinator. Similarly, the degrees of n require that each port instance of the coordinator is attached to n connectors. The behaviors of the two component types enforce that once the resource is acquired by a component of type B, it can only be released by the same component. This happens because the begin port of a B component interacts with the take port of Mutex Manager leading the latter to the control location taken. Afterwards, no other B component can fire begin, until Mutex Manager returns to the control location free ,which happens when the finish port of the former B component is fired.

Cardinalities, multiplicities and degrees may also be intervals. Let us consider, a port type p with its multiplicity defined as interval. By the interval attributes ‘sc$[x, y]$’ (single choice) and ‘mc$[x, y]$’ (multiple choice), we mean that the same (resp. a different) multiplicity is applied to each port instance of p, provided that it lies in the interval.

Chapter 3

Early validation of system requirements and design

3.1 Introduction

The design problem in systems engineering concerns with defining the architecture, modules, interfaces and data for a system, in order to meet given requirements [17]. Initially, requirements are high-level mission statements (conditions or capabilities that are also called stakeholder requirements) [18], from which system requirements are derived that define what the system must do to satisfy stakeholder requirements [19]. In this chapter, we focus specifically on system requirements; when we refer to stakeholder requirements we do so explicitly.

In [1] and [20], two perspectives of *rigorous system design* are introduced. In particular, the focus is on the design problem for systems that continuously interact with an external environment; such systems usually involve concurrent execution and emergent behaviors. The design process can be decomposed into two phases. During the first phase, which is called *proceduralization* in [1], the declarative system requirements are transformed into a procedure, i.e., a model prescribing how the anticipated functionality can be realized by executing sequences of elementary functions. During the second phase, which is called *materialization*, the procedure is implemented in a system that meets all extra-functional requirements by using available resources cost-effectively.

In this chapter, we introduce a *model-based* approach for the proceduralization phase, which aims to the systematic development of a design solution for a set of system requirements. The design problem is well-defined, only if requirements fulfill essential properties, i.e., if they are complete, consistent, correct (valid for an acceptable solution), and attainable. However, requirements provide in principle only a partial specification, which according to the current industrial practice (even for critical systems) is stated using a simplified controlled natural language (i.e. restricted in syntax and/or lexical terms). However, natural language is ambiguous [21] and it is not tied to a formal semantics. Thus, *none of the essential properties can be easily proved.*

3.1.1 Research objectives

The main objectives of our approach is to provide the means for:

- *unambiguous specification* of requirements;
- *early assurance* of consistency between the requirements and design correctness;
- use of *correct-by-construction* techniques to limit the need for a posteriori model checking.

Some of the aforementioned objectives are related to the requirements formalization challenge [22], [23] that refers to the transformation of requirements into *formal properties*. These property specifications should be implied by the system's structure and behavior in conjunction with its external stimuli [24].

We provide a systematic stepwise design approach for transforming declarative system requirements into procedures (proceduralization). This happens by incrementally building a formal and executable model of a design solution (design model), i.e., a blueprint of the system structure and behavior. The design model provides early evidence of design correctness and consistency. If the properties derived from the requirements cannot be fulfilled by the design model, a different design should be pursued or certain unsatisfied requirements have to be revised. Such an approach incurs extra cost to be paid towards delivering early evidence that the requirement specifications are realizable; on the other hand, late-stage validation, relying on testing and requiring high-cost corrective measures, can be drastically reduced.

3.1.2 Context and contributions

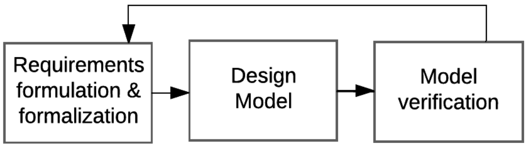


Figure 3.1: The model-based approach

Figure 3.1 outlines the proposed approach, where our research objectives are attained in three consecutive phases. In the *Requirements formulation and formalization* phase, we formulate requirements by instantiating textual templates, called *boilerplates* (like in [25]–[27]), which are filled with catalogued concepts of the system's context. The formalization of requirements as properties occurs in a semi-automated way, based on a predefined mapping of boilerplates to formal property *patterns* and a user-defined association of requirements' concepts to events of the design model. Through precisely stating how the boilerplates and concepts of requirements are transformed into properties using predefined and user-defined mappings, we achieve the unambiguous specification of requirements, since they are ensured to have a consistent interpretation with respect to the design model.

In the *Design model* building phase, the system's components are treated as blocks of established functionality; they have to be coordinated while they are progressively assembled and integrated so as to fulfil the system requirements. We adopt the main principles of [1]:

- a component-based modeling framework for enhanced productivity through reuse of model artifacts;

- the modeling language BIP, which provides an expressive component framework adequate for a semantically coherent process; any BIP model can be formally analyzed and simulated with the BIP tools[1];
- *correctness-by-construction* based on property enforcement and property composability while integrating the model components; to this end, we utilize recent theoretical results [14] together with proper automation support.

In the *Model verification* phase, we formally verify the obtained design model to check that the non-enforceable properties are fulfilled. Verification takes place, as a final step, after correct-by-construction techniques have been applied. If the properties cannot be fulfilled, a different design should be pursued or certain unsatisfied requirements have to be revised.

The relevant concrete research contributions of this chapter are:

i. The model-based process for the early validation of system requirements and design.

ii. The technical approach for the formalization of requirements. This includes the natural-like template languages for specifying requirements and formal properties, as well as the associations between templates, for the derivation of properties.

iii. A library of BIP *architectures* for simple designs [15], [28] and their associations with patterns for properties that can be enforced using our correctness-by-construction approach. These architectures were adequate to enforce all safety properties for two industrial studies through correct-by-construction model transformations.

iv. A brief account of the tool-support for the automation of the process, including a new tool called RERD (Requirements engineering for Rigorous Design).

v. A report on the early validation of requirements in two studies: the control software of the CubETH nanosatellite [15], [28], and an extract of software requirements for the telecommand management of a low orbit earth

[1] http://www-verimag.imag.fr/BIP-Tools-93

observation satellite.

Section 3.2 discusses the overview of the model-based process steps, which are thoroughly seen in Sections 3.2.1, 3.2.2, 3.2.3 and 3.2.4 together with the corresponding technical approaches. In Section 3.3, we refer to the tool-support and in Section 3.4 we provide a brief report on the results from the two case studies. The related work is surveyed in Section 3.5. Finally, the chapter concludes with a discussion on the identified benefits and limitations, as well as on the further development of our model-based process and its tool-support.

3.2 The model-based process

Any *system* under design is intended to accomplish a set of *functions* with each of them defining a stateful processing of input. The system's *functional architecture* is a top-down decomposition of its functions (using e.g. function trees [29]). The functions must fulfill certain requirement specifications, i.e. statements that delimit the problem of system design. In effect, this is only a partial specification which assumes some common and often tacit knowledge for the problem domain (domain knowledge [30]), such as physical laws for the system's external stimuli [31], standardized protocols, services and libraries.

On the side of the design solution space, a design is defined based on a hierarchical description (using e.g. product trees [32]) of the system's hardware and software *components*, known as *physical architecture*. The functions and their associated requirements are then allocated to the components of the physical architecture.

For the specification of requirements and properties, we employ two natural-like languages with precisely defined semantics. Requirements are specified using composable *boilerplates* [19], i.e., semi-complete specifications, with placeholders to be filled with concepts that adhere to a *conceptual model* of the system under design. The conceptual model encodes the relationships among the concepts used in the placeholders. With proper tool-support, the engineer avoids indeterminate references and maintains links between concepts that exist in requirements. In order to derive the properties that capture each requirement, we have mapped each boilerplate to one or more property *patterns*, that are also natural-like language templates with placeholders. These patterns associate the properties with a formal representation in a logic language.

If requirements (and derived properties) are simultaneously satisfied by the design model, then early assurance of consistency and correctness is provided (we do not cope though with inconsistencies between requirements at the specification level, which are treated e.g. in [33] and other works). The design model is incrementally built using correct-by-construction model transformations, which integrate reusable BIP architectures [14]. The integrated architectures provably discharge the specified properties through coordinating the model components. This is an automated step aiming to preserve the previously established properties. Only the properties that cannot be enforced by design need to be verified by model checking.

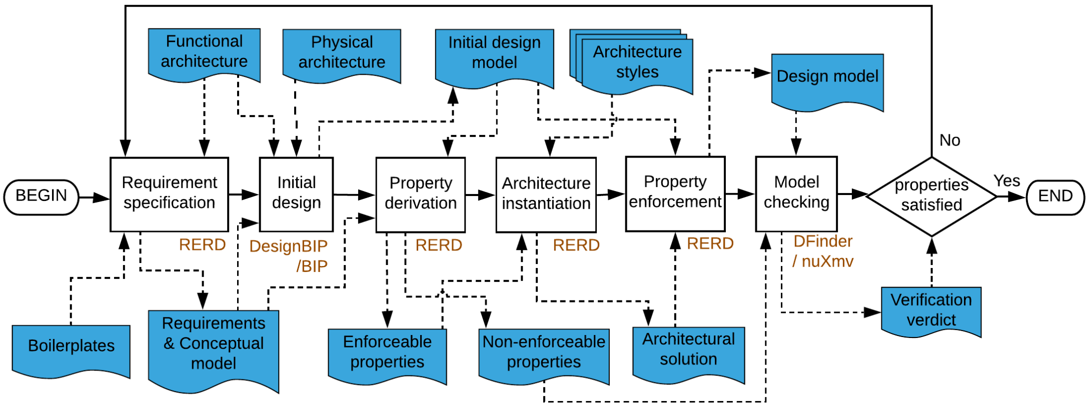


Figure 3.2: The model-based process for the formalization of requirements and design

Figure 3.2 introduces the overall process by showing the steps along with their input and output data:

Input: (i) the functional architecture

(ii) the physical architecture

Output: a design model satisfying the derived properties OR requirements that are not satisfied

Step 1 *Requirement specification:* Requirements for each function of the functional architecture are specified based on predefined boilerplates (cf. Section 3.2.1).

Step 2 *Initial design:* An initial design model is manually built with BIP components representing the physical architecture (cf. Section 3.2.2). BIP components implement behavior for the actions performed by the allocated functions; the interactions among the components encode the invocation of actions.

Step 3 *Property derivation:* Properties are derived from the specified requirements (cf. Section 3.2.3). To this end, we have associated each boilerplate with the predefined property patterns that can formally capture it. Then, for each requirement, properties are derived by filling in the patterns with elements of the design model that represent the concepts used in the boilerplate.

Step 4 *Architecture instantiation:* Properties which can be enforced by design are identified; every such property is provably enforced by a BIP architecture. The architecture to be used is instantiated (cf. Section 3.2.4) by defining the *operands* of an existing *architecture style* [10], [15], i.e., components of the design model in place of the style *parameters*, which fulfill *assumed properties*.

Step 5 *Property enforcement:* The architectures are incrementally applied to the design model (cf. Section 3.2.4) [14]. The properties assumed by definition for the operands of an architecture are verified locally by inspecting the corresponding components, before the architecture is applied to the design model. If an assumed property is not satisfieed, the component behavior will have to be refined to ensure property satisfaction.

Step 6 *Model checking:* Properties that could not be enforced using existing architecture styles are verified on the final design model. If these properties are satisfied, then so is the whole set of requirements; otherwise, the design model should be refined or certain unsatisfied requirements have to be revised.

The steps 1, 3, 4 and 5 are supported by the RERD tool, which is described in Section 3.3. The BIP design model is compiled and simulated with the BIP tools [34], whereas its deadlock freedom is checked with the D-Finder tool [8]. DesignBIP[2] [35] is a web-based graphical editor for BIP models, which can be used for the creation of the initial design model. For the verification of properties (Step 6) by model checking, it is possible to use the the nuXmv model checker [36]. Additionally, safety properties can be expressed as observer automata [37], which are then verified with the BIP tools. Three engineering roles are involved in the process, namely the Requirement Engineer for the specification of requirements (Step 1), the System Software Engineer for the system design (Steps 2, 4, 5) and the Verification Engineer for the property derivation and model checking (Steps 3, 6).

3.2.1 Requirement specification

One of the main objectives of our approach is to tackle the ambiguity of natural language requirement specifications through the use of boilerplates in combination with a conceptual model. According to [25], a boilerplate consists of *attributes* and *fixed syntax elements*, such as:

$$\langle \textit{function} \rangle \text{ shall } \langle \textit{action} \rangle$$

where "shall" is a fixed syntax element, while ⟨*function*⟩ and ⟨*action*⟩ are attributes of placeholders for user input.

Table 3.1: Conceptual classes

Class	**Definition**
⟨*function*⟩	A function of the functional architecture.
⟨*action*⟩	A processing step of a function.
⟨*state*⟩	A condition that enables/disables actions.
⟨*state-set*⟩	A set of mutually exclusive states.
⟨*event*⟩	A nominal or failure effect of an action or an external stimulus.

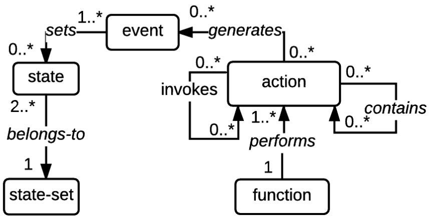


Figure 3.3: Conceptual diagram of classes.

In order to avoid indeterminate values in boilerplate attributes, we link these values with uniquely identified concepts from the conceptual model, where each

[2] https://github.com/DesignBIP/DesignBIP

concept is an instance of a *class* with precisely defined *relationships*. The conceptual classes are defined in Table 3.1 and the essential relationships for supporting the modeling steps of the process are shown in Figure 3.3. Each function *performs* actions in order to interact with other functions or the environment. In particular, actions can invoke actions of other functions or generate events. Moreover, actions are of different granularity, hence some actions are *action containers* i.e. their execution involves the execution of more fine-grained actions. Events are either generated as the effect of actions or by the environment. Specifically, an event occurs upon the end of one of its associated actions. The occurrence of ceratin events triggers a change (*set*) in the state of one or more state-sets. Notice that the diagram doesn't show two reasonable constraints for the actions, i.e., that they can invoke only actions of other functions and that they can contain only actions of their own function. Also, a constraint for the states is that they are set by events generated by actions and not by external stimuli.

Our boilerplate language is similar to the one used in [25], [38], where a boilerplate consists of at most three clauses: (i) the *prefix* clause, which specifies a stimulation or a condition, (ii) the *main* clause, which specifies an expected system action or state and (iii) the *suffix* clause, which specifies various additional constraints. Moreover, each boilerplate attribute is associated with a specific class of our conceptual model. The definition of boilerplates as a sequence of different clauses offers modularity, simplifies the problem of boilerplate definition and their interpretation using formal properties.

Example 1. *Let us consider the following natural language requirement:*

Log-001 *Every time a hardware error is detected,*
it shall be stored in a memory region in the RAM.

This requirement is expressed in active voice, using a prefix and a main clause for defining the triggering event and the system's action, respectively, as follows:

Log-001 *Prefix: If ⟨event: a hardware error is detected by a function ⟩,*
Main: ⟨function: the function ⟩ shall ⟨action: store the error in a memory region in the RAM ⟩.

△

Table 3.2: Prefix clauses

ID	Template
P1	if ⟨*event*⟩
P2	if ⟨*event*⟩ and ⟨*state*⟩
P3	while ⟨*state*⟩

Table 3.3: Main clauses

ID	Template
M1	⟨*function*⟩ shall ⟨*action*⟩
M2	⟨*function*⟩ shall ⟨*action*⟩ (and ⟨*action*⟩)+
M3	⟨*function*⟩ shall ⟨*state*⟩

Table 3.4: Suffix clauses

ID	Template
S1	before ⟨*event*⟩
S2	sequentially

Table 3.3 defines the syntax for the main clauses of our boilerplate language, whose subject is a function, that may (i) execute an *action* (M1), or (ii) execute a sequence of *actions* (M2), or (iii) be in a certain *state* (M3). The main clause is mandatory. It is the core of the requirement.

Prefixes (Table 3.2) refer to hypothetical conditions on events and/or states. They specify conditions for the main specification, i.e., for the action, the sequence of actions or the state observation mentioned in the main clause. According to the prefixes, the main clause shall occur: (i) if an *event* has occurred (P1), (ii) if an *event* has occurred and a *state* is observed (P2), or (iii) throughout an interval, where a *state* can be observed (P3). The conditions that involve events are necessary and sufficient, while those consisting only of states simply represent a necessity.

A suffix is used to constrain the main specification. The suffix clauses shown in Table 3.4 specify that each time the main specification (action, sequence of actions or state observation) is activated, it shall: (i) have ended before an *event* occurs (S1), or (ii) occur sequentially (i.e., consecutive activations do not overlap in time) (S2)

Let us consider the boilerplate consisting of the P1, M1 and S2 templates, specifying that "if *event*, *function* shall *action* sequentially". Such a boilerplate expresses that: (i) *event* is a necessary and sufficient precondition for one action occurrence and (ii) consecutive *action* occurrences are constrained to be executed sequentially. The remaining prefix-suffix combinations are interpreted accordingly.

During the specification of each requirement, the conceptual model is enriched with new concepts, if the existing concepts are not sufficient. At the end of the specification step, the conceptual model will contain the concepts used in the requirements and *additional* concepts that are related to them. For example, events used in the requirements will be related to their generating actions, even if these actions are not explicitly mentioned in requirements. The conceptual model's quality is a responsibility of the Requirement Engineer. This matter has been examined in related works [39], [40] that are further discussed in Section 3.6.

Example 2. *Let us consider the requirements in Table 3.5, which have been defined for the function that handles the housekeeping of the payload (PL) subsystem (abbreviated as HK PL). The concepts in requirements and other concepts related to them are depicted in the conceptual model of Figure 3.4, which shows that:*

Table 3.5: Requirements for the *HK PL* function

ID	Requirement
HK-02	P2: if ⟨event-e003: [TBD] sec pass ⟩ and ⟨state-s003: HK collection is enabled for PL ⟩ M1: ⟨function: HK PL ⟩ shall ⟨action-a004: handle HK data from the PL ⟩
HK-03	P3: if ⟨state-s002: PS[3] for PL is not enabled ⟩ M1: ⟨function: HK PL ⟩ shall ⟨action-a002: transmit HK data through the TC/TM service ⟩
HK-04	P3: while ⟨state-s001: PS for PL is enabled ⟩ M1: ⟨function: HK PL ⟩ shall ⟨action-a001: write HK data to the flash memory ⟩
HK-05	P1: if ⟨event-e004: a PL failure persists for [TBD] sec ⟩ M1: ⟨function: HK PL ⟩ shall ⟨action-a003: contact the EPS for a restart of the PL ⟩

- *states s001 and s002 belong to the state-set st001, thus, only one of them can be observed at a given instant. Each of these states is set by the events e001 and e002, respectively (states s003 and s004 are similarly related).*

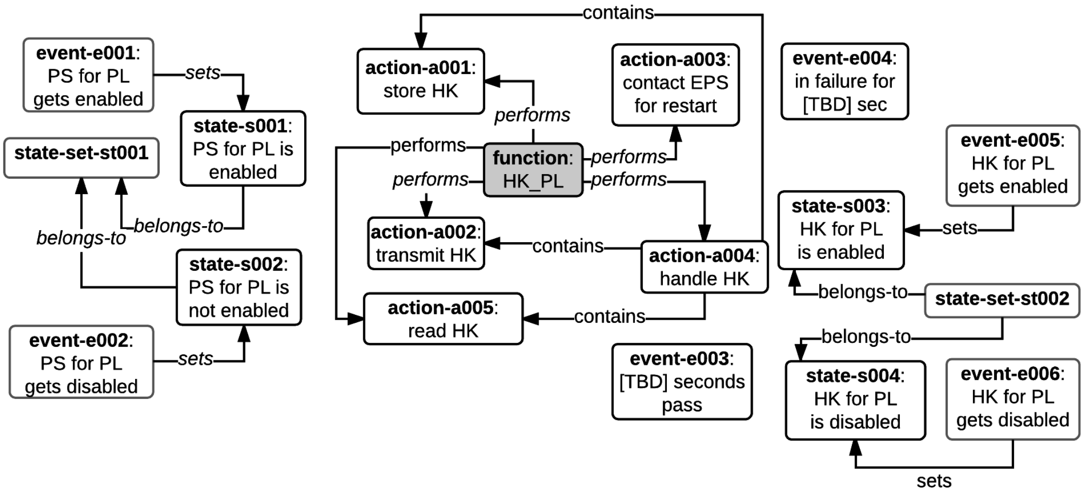

event-e001: PS for PL gets enabled
sets
state-s001: PS for PL is enabled
state-set-st001
belongs-to
belongs-to
state-s002: PS for PL is not enabled
event-e002: PS for PL gets disabled
sets
contains
action-a001: store HK
performs
performs
function: HK_PL
performs
performs
performs
action-a002: transmit HK
contains
action-a003: contact EPS for restart
action-a004: handle HK
action-a005: read HK
contains
event-e003: [TBD] seconds pass
event-e004: in failure for [TBD] sec
event-e005: HK for PL gets enabled
sets
state-s003: HK for PL is enabled
belongs-to
state-set-st002
belongs-to
state-s004: HK for PL is disabled
event-e006: HK for PL gets disabled
sets

Figure 3.4: Conceptual model for the requirements of the *HK PL* function

- *the used action a004 represents an action container that consists of a001, a002 and a005.*
- *events e003 and e004 are neither generated by an action nor do they set any states.*

For brevity, Figure 3.4 omits the invokes relationships that relate these actions to actions of other functions. These relationships are shown at later steps of the running example.

△

The templates in Tables 3.2, 3.3 and 3.4 in no way form a complete set of boilerplates adequate for all kinds of system requirements, since the boilerplate language is not the primary goal of this thesis. Thus, our prefixes can only express necessary and sufficient conditions based on one state or event, even though requirements are often subjected to more complex conditions (e.g. based on two events) or to conditions that are either necessary or sufficient. However, we opted to keep the boilerplate language simple enough for illustrating the main principles behind its design, while covering the specification needs of the two case studies in Section 3.4. Our considerations for the evolution of the current language are discussed in Section 3.6.

3.2.2 Initial design

The initial design step generates the design model in its initial form, which is a manually built blueprint of the system's functional behavior. All the concepts of actions and events mentioned in the requirements should be traceable in ports of the initial design model.

The model consists of BIP components that implement functions of the functional architecture. Each *action* of the conceptual model, which is an identifiable block of functionality within a function, is represented by a list of ports of a component. Events that are generated by actions are also represented by the action's ports, whereas environmental events are non-deterministic inputs which are not explicitly modeled. Components may enclose one or more atomic subcomponents in order to enable ports within separate threads of control. The number of atomic components to be used and the placement of actions is a design choice that depends on possible order dependencies among the actions. For instance, actions which are executed alternatively should be enabled at the same control location of a component, whereas actions that are independent with each other should be placed in different components.

The invocation of actions, which is reflected by the "invokes" relationship in the conceptual model, is represented by component interactions. Separate interactions are included for issuing an invocation and receiving the output. Rendezvous connectors can model *synchronous* invocations, where the caller has to wait for the output. For *asynchronous* invocations, an additional atomic component should be used for buffering the output before the caller can get it. Actions may return a nominal output or possible failures. The caller may receive all outputs with the same port or using different ports, if it needs to distinguish among them (e.g. if it should be transferred to different control locations).

The design choices at this step incur a limited complexity and risk to the whole process, since components of the initial design model should have elementary behaviors. More complex behaviors are only built with architecture instantiation in a controlled and rigorous way.

Example 3. *The initial design model shown in Figure 3.5 corresponds to the requirements in Table 3.5. It includes the following three components of the physical architecture:*

- *the* `HK PL`*, which handles the Housekeeping for the PL subsystem function;*
- *the* `I2C_sat`*, which handles the communication through the I2C bus [41] function;*
- *the* `Flash Memory`*, which handles the flash memory data management function.*

Figure 3.4 shows the actions of the function allocated to the `HK PL` *component. The other two components are included in the model since their actions are invoked by* `HK PL`*. The* `HK PL` *actions have been placed into two atomic subcomponents of the HK PL, namely the* `HK PL read` *, which reads Housekeeping data, and the* `HK PL restart`*, which activates a restart of the* `PL` *subsystem. Actions are mapped to lists of ports as follows:*

a001 → [`mem_write_req` *,* `mem_res`*]*
a002 → [`I2C_ask_TTC` *,* `I2C_res_TTC`*]*
a003 → [`I2C_ask_EPS` *,* `I2C_res_EPS`*]*
a004 → [`beginHK` *,* `finished`*]*
a005 → [`I2C_ask_PL`*,* `I2C_res_PL` *,* `I2C_fail_PL`*]*

The use of two atomic components is driven by existing dependencies among actions. For example, in `HK PL read`*, the action of reading housekeeping data (a004) should precede their transmission (a002) or storage (a001). On the other hand, subsystem's reset (a0005) occurs independently of other actions.*

In Figure 3.5 a simplified presentation of BIP connectors is shown by using the diamond shapes in component interfaces. Each diamond is attached with ports that participate in one action's invocation and the receipt of the result/failures and the link between two diamonds denotes that BIP connectors exist between these ports. All actions of the `HK PL` *function invoke actions of the* `I2C_sat` *and* `Flash Memory` *components*[4]*. Specifically, the action of subsystem communication of the* `I2C_sat` *is invoked by three actions that need to contact other subsystems:*

[4] the *invokes* relationship is not shown in the conceptual model of Figure 3.4

- *a004, which reads housekeeping data from the PL subsystem;*
- *a002, which submits data to the TC subsystem for transmission to the ground;*
- *a005, which contacts the EPS subsystem for the restart of the PL subsystem.*

Moreover, the memory write action of the `Flash Memory` *is invoked by a003 for writing the data to the flash memory storage. Note here that a failure in reading the housekeeping data from PL leads to a different control location than the nominal output and it is therefore received by a different port (i.e.* `I2C_fail_PL`*) than the port receiving the nominal output (i.e.* `I2C_res_PL`*). In contrast, both outputs of the memory write action can be received with the* `mem_res` *port, since they lead to the same control location.*

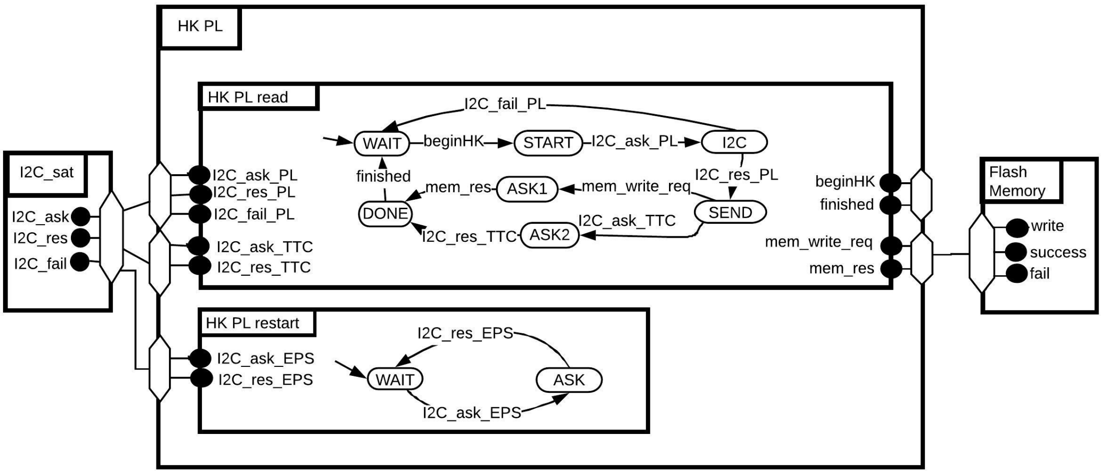


Figure 3.5: Example of an initial design model

△

3.2.3 Property derivation

Formal properties are bound to a unique interpretation specified in an analyzable language. For behavioral and architectural properties of design models in BIP, we usually use the Computational Tree Logic (CTL) [42] and the configuration logics [10], respectively. However, since we aim at a general approach for specifying properties, we use the specification framework of [43] with patterns that are formally defined in CTL and in other languages[5]. These patterns

[5] http://patterns.projects.cs.ksu.edu/

have been found expressively sufficient to capture requirements written with our boilerplates.

Each property specification consists of two templates, a *scope* and a *pattern*. The pattern defines an expected occurrence or the order of one or more events. The scope selects the subset of the model state-space, where the pattern is expected to hold true. For the rest of the state-space, the property is undefined. For the set of our boilerplates, it suffices to derive properties using the *existence*, *absense*, *precedence* and *response* patterns of [43]. Also we needed two scopes, namely the *Global* scope or the *Between...And* scope that refers to a part of the state-space. The templates for the patterns and the scopes are shown in Table 3.6. Their placeholders are filled with logical propositions (*beh*), that are specified as follows:

- Atomic propositions are defined over firings of component ports: a port p of a component A is denoted by $A.p$ and holds true at a global state in which the port has fired.
- Logical connectives & (and), | (or) combine atomic propositions with their usual meaning.
- Temporal modalities are used to build more complex propositions. In particular, with the *next* operator ($\mathbf{X}$) in front of a *beh*, we refer to the next global state after *beh* occurs (the next operator can be formally expressed in CTL as $\mathbf{AX}$).

Table 3.6: Templates for scopes and patterns

ID	**Template**	**Description**
Global	globally,	throughout the whole execution
Between...And	between $\langle beh \rangle$ and $\langle beh \rangle$,	from a $\langle beh \rangle$ to another $\langle beh \rangle$
Existence	$\langle beh \rangle$ exists	a $\langle beh \rangle$ is observed
Absense	$\langle beh \rangle$ is absent	a $\langle beh \rangle$ is not observed
Precedence	$\langle beh \rangle$ precedes $\langle beh \rangle$	a $\langle beh \rangle$ is observed before another $\langle beh \rangle$
Response	$\langle beh \rangle$ responds to $\langle beh \rangle$	a $\langle beh \rangle$ is observed after another $\langle beh \rangle$

We derive properties from requirements, based on a mapping from the requirement's boilerplate to combinations of scope and pattern templates that are shortly referred to as "property patterns". This association, which is shown in Table 3.7, refers to a set of symbols, which map the boilerplate attributes to *beh* propositions. The mappings have to be manually created by the System Software Engineer, as follows:

- The *beg* and *end* symbols map actions to the *beh* propositions that define their beginning (resp. ending). For instance, the beginning of an action is the port with which it can be invoked and its ending is the port of sending its response or a disjunction of ports (e.g. when alternative endings exist).
- The *occ* symbols map events to *beh* propositions that define each event's occurrence. An internal event is generated by one or more action(s), hence a *beh* is the disjunction of *end* symbols of alternative actions generating the event. The occurrence of an external event is a port that generates external stimuli. Such ports are not part of the initial model; instead, we consider them "virtual ports" of a "virtual component" named `Environment`, in order to assign them to *occ* symbols in property derivation.
- The *obs* symbols map states to ports, which are enabled when the design model *is* in each particular state. These ports are not part of the initial model; instead, they are placed in coordinating components of architectures that are added during property enforcement. Hence, we consider them "virtual ports" in property derivation.

In addition to the aforementioned symbols, the *beg(M)* and *end(M)* symbols (see the footnote ^{b} in Table 3.7) are automatically evaluated based on the used main clause template.

The semantics for M1 and M3 templates, alone, do not yield any correctness properties. On the other hand, M2 specifies a sequential execution of N actions, which is expressed by the conjunction of properties, $M2.1.i$ (see Table 3.7), defined for each action a[*i*] in the sequence (except for the last one). The property expresses that:

- the end of action a[i] enables the beginning of action a[i+1], i.e., "*globally, a[i] should end before a consecutive beginning of a[i+1]*", formulated as:

$M2.1.i$: globally, *end*(a[i]) precedes *beg*(a[i+1]).

Another example is the P2 prefix template, from which patterns *P2.1* and *P2.2* are derived. The patterns express that:

- the observation of an event while being in a state enables *beg*(M), i.e., "*globally, the event and the state are observed at some time instant before beg(M)*", formulated as:

  P2.1: globally, *obs*(e1) ∧ *obs*(s1) precedes *beg*(M)

- the observation of an event while being in a state triggers *beg*(M), i.e.,"*globally, beg(M)* follows the observation of the event and the state at some time instant", formulated as:

  P2.2: globally, *beg*(M) responds to *occ*(e1) ∧ *obs*(s1)

The rationale of the other derived properties is discussed in A.1.

Table 3.7: Boilerplate templates and their associated patterns

Boilerplates	**Derived patterns**
P1: if *e1*, ... [a]	*P1.1:* globally, *occ*(e1) precedes *beg*(M) [b] *P1.2:* globally, *beg*(M) responds to *occ*(e1)
P2: if *e1* and *s1*, ...	*P2.1:* globally, *occ*(e1) ∧ *obs*(s1) precedes *beg*(M) *P2.2:* globally, *beg*(M) responds to *occ*(e1) ∧ *obs*(s1)
P3: while *s1*, ...	*P3.1:* between *beg*(M) and **X** *beg*(M),*obs*(s1) exists
M1: f1 shall *a1*	-
M2: f1 shall *a1* and ... and *aN*	*M2.1.i:* globally, *end*(a[i]) precedes *beg*(a[i+1])
M3: f1 shall *s2*	-
S1: ... before *e2*	*S1.1:* between *obs*(P) and *beg*(M), *occ*(e2) is absent [c]
S2: ... sequentially	*S2.1:* between *beg*(M) and *beg*(M), *end(M)* exists

[a] The enumerated *fi*, *ai*, *ei* and *si* denote a function, action, event and state mentioned in the requirement.

[b] *beg*(*M*) and *end*(*M*) are replaced according to the used main clause *M* as follows:

$$\mathit{beg}(M)=\begin{cases}\mathit{beg}(a1) & \text{if M=M1 or M=M2}\\ \mathit{obs}(s2) & \text{if M=M3}\end{cases}\quad \mathit{end}(M)=\begin{cases}\mathit{end}(a1) & \text{if M=M1}\\ \mathit{end}(aN) & \text{if M=M2}\\ \neg\mathit{obs}(s2) & \text{if M=M3}\end{cases}$$

[c] *obs*(*P*) is replaced according to the used prefix *P* as follows:

$$\mathit{obs}(P)=\begin{cases}\mathit{occ}(e1) & \text{if P=P1}\\ \mathit{occ}(e1)\wedge\mathit{obs}(s1) & \text{if P=P2}\\ \mathit{obs}(s2) & \text{if P=P3}\end{cases}$$

Example 4. *Let us consider the requirement HK-02 of our running example, which is captured by the P2.1 and P2.2 property patterns. For these patterns, the following symbols have to be assigned with ports:*

- *occ(e1), is assigned with the "virtual port"* `Environment.HKPL_TBDpass` *port modeling the occurrence of the external event* $e1$*;*
- *obs(s1) is assigned with the "virtual port"* `HK_PL.enabledHK_PL` *modeling the observation of the state* $s1$*;*
- *beg(a1) is assigned with the* `HK_PL.beginHK` *port modeling the begining of action* $a1$.

△

3.2.4 Architecture instantiation and property enforcement

Nine architecture styles from those introduced in [15], [28] were adequate to satisfy the safety properties of our case studies. In this section, we outline how property enforcement is achieved using four out of these nine styles, namely:

- the *Action flow*, which enforces an ordering of actions;
- the *Mode management*, which restricts the set of actions performed in a mode (state);
- the *Event monitoring*, which reports upon monitored events;
- the *Mutual exclusion management*, which ensures mutually exclusive access to a critical section.

While these styles represent recurring patterns of satellite on-board software, we believe that they are not tied to the given problem domain.

In order to apply an architecture, the architecture style's parameters have to be defined. Then, the architecture is instantiated and combined with other architectures that have already been applied to the same operand components (using the $\oplus$ operator as described in Section 2.2). In our design process this is an automated step, which merges the connectors of architectures applied on common ports. The result of applying multiple architectures to the design model has to be verified for deadlock-freedom.

Action flow

The Action flow architecture style, shown in Figure 3.6, enforces a sequential flow on N actions allocated to n components of type B, using an Action Flow Manager coordinator component. Assuming that n_a actions of the flow belong to one component, the component has n_a instances of the actBegin and actEnd port types, which represent the beginning and end of each action. The coordinator resets the action flow only after the N-th action has ended. Connector degrees imply that each action can only be involved in one action flow.

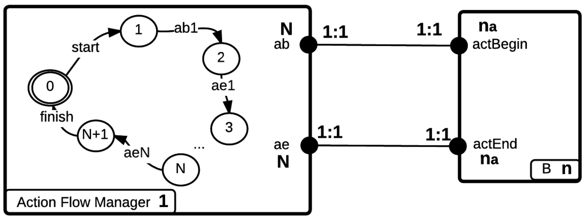


Figure 3.6: Architecture diagram of the Action flow style

The Action flow style is used to enforce a collection of properties of the M2.1.i pattern ($i = 1, \ldots, N$) derived from the same requirement. Such patterns specify that, given a set of actions $a[1] \ldots a[N]$, *the end of action a[i] enables the beginning of a[i+1].* For each action a[i], the port instances that should be mapped to each actBegin[i] (resp. actEnd[i]) are the port(s) that correspond to the *beg*(a[i]) (resp. *end*(a[i])):

$$\texttt{actBegin[i]} \rightarrow beg(a[i])$$

$$\texttt{actEnd[i]} \rightarrow end(a[i])$$

Example 5. *Let us consider the requirement CDMS-02 of the CubETH case study:*

P1: ⟨e1: if [TBD] seconds pass ⟩

M2: ⟨f1: CDMS_status ⟩ shall ⟨a1: reset the internal and external watchdogs ⟩ and ⟨a2: contact the EPS subsystem with a "heartbeat" ⟩

from which the following property of the M.2.1 pattern is derived:

CDMS-02-M.2.1: globally, end(a1) precedes beg(a2)

Let us assume that actions a1 and a2 are placed in the `Watchdog reset` *and the* `Heartbeat` *components, respectively. For the enforcement of the property, an Action flow architecture was instantiated using the two components as operands of type* `B`*. Table 3.8 shows the mapping of their ports for actions a[1] and a[2] to port type parameters. Figure 3.7 presents the result of applying the architecture, which adds the coordinator and two connectors shown with dashed lines. Since the coordinator represents the* `Heartbeat` *component (i.e., all its ports are synchronized with ports of the coordinator), the latter is removed as redundant. Moreover, any symbols that refer to the removed component's ports are updated to refer to the ports of the the coordinator.*

Table 3.8: Action flow architecture style parameters

	a**[1]**	a**[2]**
actBegin	`Watchdog_reset.internal_watchdog`	`Heartbeat.send`
actEnd	`Watchdog_reset.done`	`Heartbeat.res , Heartbeat.fail`

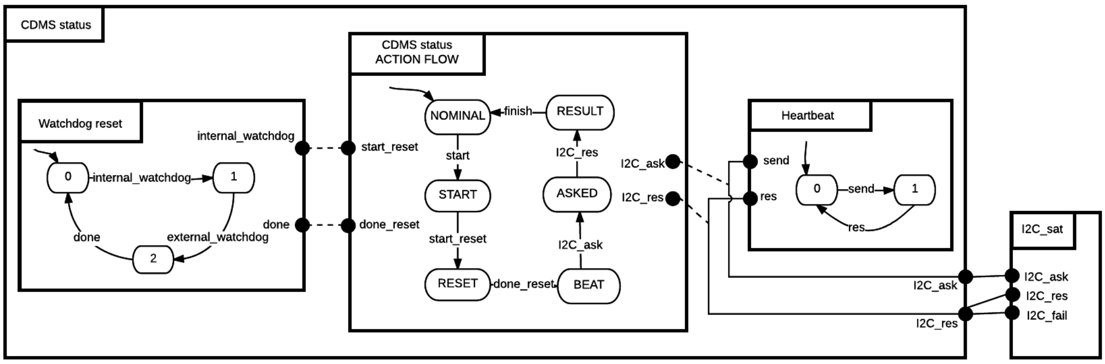


Figure 3.7: Application of an Action flow architecture

△

Mode management

The *Mode management* architecture style (Figure 3.8) restricts the set of actions which can be executed (i.e., enabled actions) based on a set of modes. It consists of one coordinator of type `Mode Manager`, n parameter components of type `B1` and k parameter components of type `B2`. Each `B2` component *triggers* the transition of the `Mode Manager` to a specific mode. `B1` components have actions that should be enabled in specific mode(s) of the `Mode Manager`. `Mode`

Manager has one control location for each mode, one port type toMode with cardinality k and k port types inMode with cardinality 1. Each toMode port is connected with the changeMode port of a dedicated B2 component.

B1 has k port types modeBegin with cardinality $mc[0,1]$. In other words, a component instance of B1 might have any number of port instances of modeBegin from 0 to k. B1 has also a modeEnd port type with cardinality k. m[*i*]b stands for "mode i begin" and indicates that an action that is enabled in mode i has begun its execution. The m[*i*]e ports stand for "mode i end" and indicate that an action that is enabled in mode i has ended. Such ports are exported as modeEnd in the interface of the B1 components. Each inMode port instance of the Mode Manager must be connected with the corresponding m[*i*]b port instances of all B1 components through an n-ary connector, where a different multiplicity in the interval $[1,n]$ is considered for each port instance.

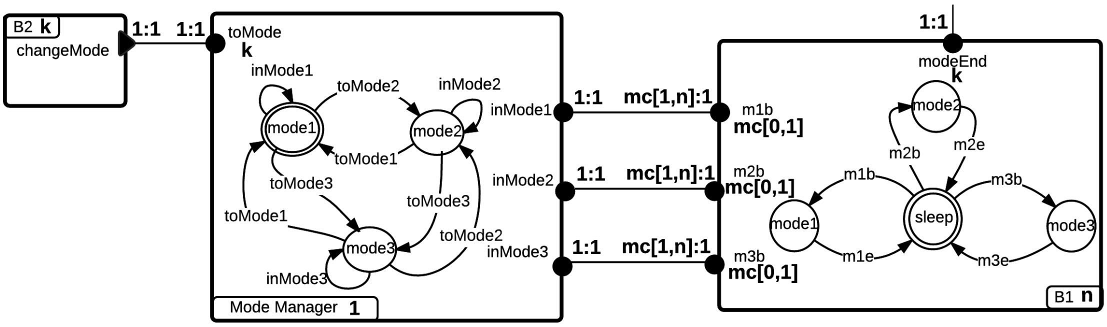


Figure 3.8: Architecture diagram of the Mode management style (component behavior is shown for k=3)

This architecture style enforces sets of properties of the P3.1 pattern that refer to states of the same state-set. According to each such property, "*the main specification shall begin only if a state is observed*". The style is parameterized by setting k equal to the number of states in the state-set. To identify instances of *m[i]b* ports, we use a new symbol enforce_beg(M), which is evaluated as follows:

- if $M = M1$ or $M = M2$, enforce_beg(M)=beg(M)
- if $M = M3$, in which case beg(M)=obs(s2), enforce_beg(M) is the *beg* of each action that triggers state $s2$; these actions are found in conceptual model by backward tracing the relationships action$\xrightarrow{generates}$event$\xrightarrow{sets}$state.

The second evaluation case reflects that the restriction of being in a state can only be ensured by restricting the event of entering in that state. Operands of type `B1` are the components having the ports mapped to the *m[i]b* ports. The `changeMode` port type is mapped to the ports of the *occ* of each event that sets the state. Operands of type `B2` are the components having these ports. After having applied a mode management architecture, each "virtual port" assigned to the *obs* of the represented states is replaced by an *inMode*[*i*] port of the `Mode Manager`.

The Mode management style is also used in combination with the Event monitoring style to enforce the P2.1 pattern. Specifically, we apply the Mode management after having applied the Event monitoring, by mapping the m[*i*]b port type to the port of the event monitoring coordinator that observes the event.

Example 6. *Let us consider the requirements HK-03 and HK-04 in Table 3.5, from which the following two properties are respectively derived:*

HK-03-P3.1: globally, obs(s002) precedes beg(a002)

HK-04-P3.1: globally, obs(s001) precedes beg(a001)

States s001 and s002 belong to the same state-set, hence, they can be enforced through a single Mode management architecture in which k = 2*. The style parameters shown in Table 3.9 associate state s001 with mode[1] and state s002 with mode[2]. The* `m[1]b` *and* `m[2]b` *port types are mapped to the beg(M) of each pattern, namely the beg(a001) (evaluated as* `HK_PL_read.mem_write_req`*) and beg(a002). Since Figure 3.4 shows that each mode is set by the events e001 and e002, respectively, the* `changeMode` *port type is mapped to the ports assigned to the occ(e001) (evaluated as* `s15_1.PL`*) and the occ(e002) (evaluated as* `s15_2.PL`*). The result of architecture application is presented in Figure 3.9, where the added connectors are shown with dashed lines.*

	mode[1]	**mode[2]**
changeMode	s15_1.PL	s15_2.PL
m[*i*]b	HK_PL_read.mem_write_req	HK_PL_read.I2C_ask_TTC

Table 3.9: Mode management architecture style parameters

△

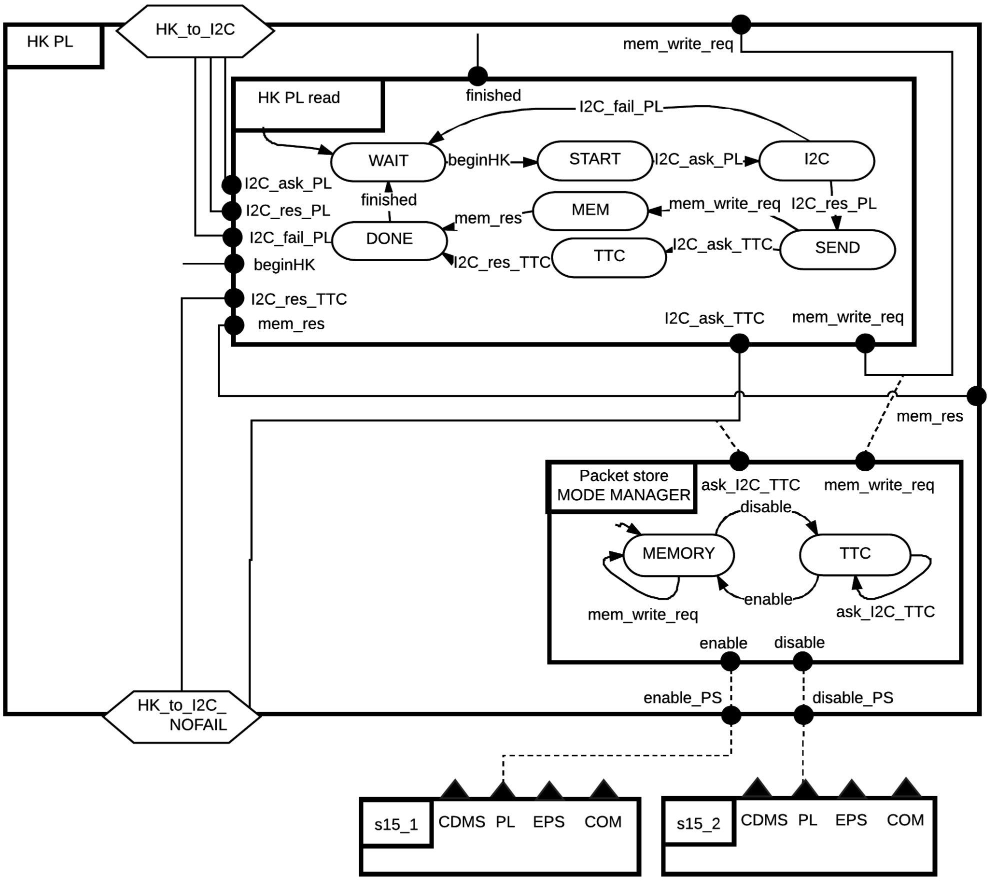


Figure 3.9: Application of a Mode management architecture

Event monitoring

The *Event monitoring* architecture style, shown in Figure 3.10, provides a coordinator component of type `Event Monitor` that tracks events of *n* components of type *B* and reports them to a component of type *service*. Each *B* component has an instance of the *sndEvent* port type, while the *service* component has an instance of the *getRep* port type.

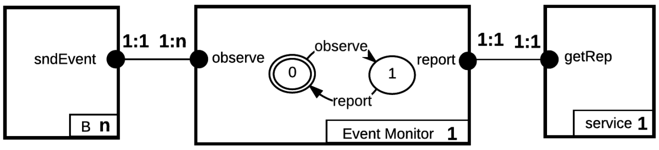


Figure 3.10: Architecture diagram of the Event monitoring style

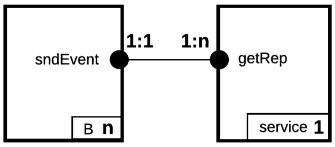


Figure 3.11: Architecture diagram of the bipartite connectors' simplification

The event monitoring architecture style is used to enforce the P1.1 and P2.1 patterns, according to which "*the main specification shall begin only if a certain* *event occurs*". For each such pattern, a separate architecture is applied, where the `getRep` port type is mapped to the ports assigned to *enforc_beg*(M) and the *sndEvent* port type is mapped to the set of ports given in the *occ* of the event. Moreover, the P2.1 pattern requires the additional application of a mode management architecture, as it has been already explained in Section 3.2.4.

Under the assumption that the action is enabled whenever the event is observed, the coordinator's behavior is reduced to a single control location and the transitions `observe`, `report` are seen as indivisible (replaced by a single port). For simplicity, the coordinator is omitted, and it is replaced by bipartite rendezvous connectors between the port(s) of the event occurrence and the action's beginning. Figure 3.11 shows the architecture diagram of the bipartite connectors' simplification.

Example 7. *Let us consider the requirement HK-01 in Table 3.5, from which the following property is derived:*

HK-02-P2.1: globally, occ(e003) $\wedge$ *obs(s003) precedes beg(a004)*

The property is enforced through a combination of an event monitoring and a mode management architecture, but here we focus on the event monitoring. The used parameters are shown in Table 3.10. The `getRep` *port is mapped to the enforc_beg(M), namely the beg(a004) (evaluated as* `HK_PL_read.beginHK`*). The* `sndEvent` *is mapped to the occ(e003) (evaluated as* `Environment.HKPL_TBDpass`*).*

`sndEvent`	`Environment.HKPL_TBDpass`
`getRep`	`HK_PL_read.beginHK`

Table 3.10: Event monitoring architecture style parameters

In this example, the event represented by the port `Environment.HKPL_TBDpass` *can be reported anytime after a deadline expires. Hence, the assumption that the reporting action is enabled whenever the event occurs is true and the bipartite connector simplification is used without affecting event occurrences (Figure 3.12).*

△

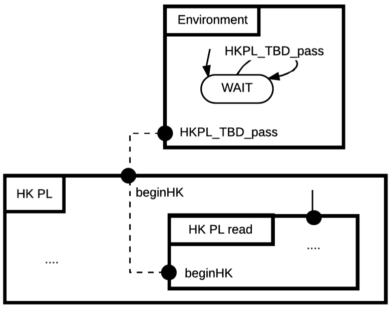


Figure 3.12: Application of the bipartite connectors' simplification of the Event monitoring architecture

Mutual exclusion management

The *Mutual exclusion management* architecture style, shown in Figure 2.4, has a coordinator component of type `Mutex Manager`, which ensures that the actions of *n* parameter components of type `B` are executed in a mutually exclusive manner. The beginning and end of actions are represented by the *begin* and *finish* port types of `B` components.

Mutual exclusion management is used to enforce the *S2.1* pattern, according to which "*consecutive executions of the main specification occur in a sequential manner*". The style is parameterized by mapping the `begin` and `finish` ports to the set of ports in *enforc_beg*(M) and the set of ports in *end*(M).

Example 8. *A mutual exclusion architecture applied to the* `Flash Memory` *component is used to enforce that the read and write requests should be processed in a mutually exclusive manner. The parameters for the architecture are those in Table 3.11. The* `begin` *is mapped to the ports for the invocation of a read/write request and the* `finish` *is mapped to their results. The obtained model is shown in Figure 3.13.*

begin	`Flash_Memory.read`, `Flash_Memory.write`
finish	`Flash_Memory.return`, `Flash_Memory.fail`

Table 3.11: Mutual exclusion management architecture style parameters

△

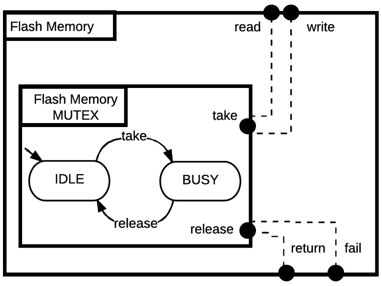


Figure 3.13: Application of a Mutual exclusion management architecture

Liveness

In general, the enforcement of liveness properties requires additional assumptions of fair execution scheduling. Furthermore, in order to guarantee the preservation of liveness properties by architecture composition, one has to verify the architectures' pair-wise non-interference [14].

However, liveness properties of the patterns P1.2 and P2.2, can, indeed, be enforced by the bipartite connectors' simplification of the Event monitoring architecture style. Let us consider the safety property "*the main specification begins atomically upon the occurrence of event e*", formulated as follows:

P1.2': between *occ*(e) and **X** *occ*(e), *beg*(M) exists.[6]

It can be easily shown that P1.2 is implied by P1.2', which can be enforced by the bipartite connectors' simplification if the assumptions for its application hold (cf Section 3.2.4).

Another way to *indirectly* enforce P1.2 through the Event monitoring architecture style is by considering the following safety property: "*after an occurrence of event e, another such event does not occur before the beginning of the main specification*", formulated as follows:

P1.2": between *occ*(e) and *occ*(e), *beg*(M) exists.[7]

It can be easily shown that P1.2", which is enforceable by the Event monitoring architecture style, implies P1.2, if it can be verified or assumed that *occ*(e) occurs infinitely often:

[6] The semantics of this property in CTL is given by the formula $\mathtt{AG}\ (occ(e) \rightarrow beg(M)])$.

[7] The semantics of this property in CTL is given by the formula $\mathtt{AG}\ (occ(e) \rightarrow \mathtt{AX}\ \mathtt{A}[\neg occ(e)\ \mathtt{W}\ beg(M)])$.

P1.2.asm: globally, *occ*(e) responds to *occ*(e)

Decision flows for property enforcement

Finding the suitable approach for enforcing a given property involves a decision-making process. Algorithm 1 introduces such a process for properties of the P1.1 pattern. The first conditional (line 1) checks whether the bipartite connector simplification can be applied, the second conditional (line 3) checks whether Event monitoring is necessary, and the else statement (line 6) is reached if the property should be verified, through inspection or model checking.

```
Data: occ(e), beg(M)
Result: P1.1 is either enforced or should be verified
if occ(e) is allocated to a different atomic component than beg(M) then
    P1.1 is enforced by the bipartite connectors' simplification of the Event
      monitoring style;
else if occ(e) is allocated to the same atomic component with beg(M) and
  P1.1 does not hold by inspection then
    P1.1 is enforced by the Event monitoring style;
else
    P1.1 should be verified
```

Algorithm 1: Decision-making process for the *P1.1* pattern

The direct or indirect enforcement of the P1.2 pattern is guided by the process shown in Algorithm 2. The flow takes into account the architecture that enforces *P1.1*, if such an architecture has been applied. The decision of the flow is either that the P1.2 property has been enforced by the architecture, or that it has to be verified through model checking. Similar processes are followed for the remaining patterns.

```
Data: the applied architecture that enforces P1.1, if any
Result: P1.2 is either enforced or has to be verified through model
        checking
if the Event monitoring style has been applied and the P1.2.asm is verified
 then
    P1.2 is enforced by the Event monitoring style;
else if the bipartite connectors' simplification has been applied then
    P1.2 is enforced by the the bipartite connectors' simplification;
else
    P1.2 has to be verified through model checking
```

Algorithm 2: Decision-making process for the *P1.2* pattern

3.3 Tool support

The RERD tool supports the requirement specification, property derivation, architecture instantiation and property enforcement, i.e. the steps 1, 3, 4 and 5 of the model-based process, whereas in step 6 the D-Finder tool is used and the nuXmv model checker [36], if there is need for verifying CTL properties. For step 1, the Requirements Engineer selects among the predefined boilerplate clauses and then inserts in each placeholder a textual description referring to a uniquely identified concept. The concept can be selected from the previously defined concepts (search support is provided) or if a new concept is needed it is entered along with its relationships. The conceptual model is stored, shared and is accessed through an underlying ontology architecture, whose design does not need to be known to the Requirement Engineer (the concept classes in Figure 3.3 suffice for specifying requirements).

Figure 3.14 shows the Requirement Editing screen of the RERD tool. The upper part of the screen allows selecting among the available boilerplate clauses, which are displayed in separate tables. In the middle part, requirements are shown in an editable form, that is, their placeholders and additional information for the requirement (e.g. id, category) can be filled in this panel. The lower part of the screen is used for browsing and searching requirements that match string(s) given in a search box. The table displays the requirements returned by each search (all requirements match an empty string), with buttons attached to each row for editing/deleting them.

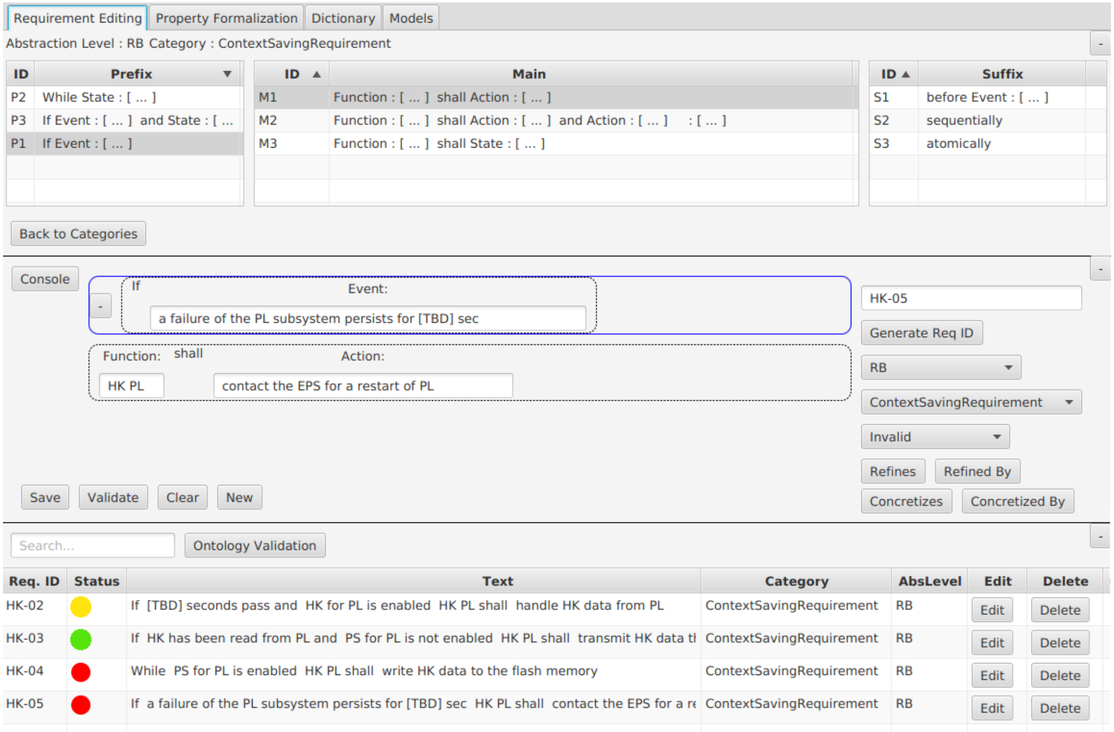
The RERD tool also stores the user-defined values for the symbols used in patterns. Specifically, the System Software Engineer assigns ports to the symbols that are necessary for the properties of the specified requirements. These symbols may be reused in more than one property. Hence, when the Verification Engineer uses the tool during the property derivation (step 3), the necessary properties are automatically created by retrieving the values of symbols.

For architecture instantiation and property enforcement (steps 4 and 5), the System Software Engineer can choose among the available architecture styles and parameterize them for creating architectures that enforce a set of properties. The architectures are then automatically applied to their operand components and the design model is updated as appropriate.

`DesignBIP` [35] is a web-based graphical editing tool, which can be used for the specification of BIP models and BIP architectures. The tool can assist the creation of the initial design model in step 1. Moreover, it allows for the creation of new architecture styles to be integrated in the RERD tool, whenever RERD is extended with new boilerplates (and enforcement opportunities).

The `D-Finder` tool [8] is used by the Verification Engineer for verifying the deadlock-freedom of the design model (step 6). `D-Finder` is capable of analyzing very large BIP models using compositional verification on an over-approximated set of reachable states. For model checking CTL properties, the BIP model has to be transformed with the BIP-to-NuSMV tool[8] into the input language of the nuXmv model checker.

[8] The tool is available from `http://risd.epfl.ch/bip2nusmv`. It is based on the encoding presented and proven correct in [44, Section 4].



Abstraction Level : RB Category : ContextSavingRequirement

ID	Prefix
P2	While State : [...]
P3	If Event : [...] and State : [...
P1	If Event : [...]

ID	Main
M1	Function : [...] shall Action : [...]
M2	Function : [...] shall Action : [...] and Action : [...] : [...]
M3	Function : [...] shall State : [...]

ID	Suffix
S1	before Event : [...]
S2	sequentially
S3	atomically

If Event: a failure of the PL subsystem persists for [TBD] sec

Function: HK PL shall Action: contact the EPS for a restart of PL

HK-05

RB

ContextSavingRequirement

Invalid

Req. ID	Status	Text	Category	AbsLevel	Edit	Delete
HK-02	🟡	If [TBD] seconds pass and HK for PL is enabled HK PL shall handle HK data from PL	ContextSavingRequirement	RB	Edit	Delete
HK-03	🟢	If HK has been read from PL and PS for PL is not enabled HK PL shall transmit HK data tl	ContextSavingRequirement	RB	Edit	Delete
HK-04	🔴	While PS for PL is enabled HK PL shall write HK data to the flash memory	ContextSavingRequirement	RB	Edit	Delete
HK-05	🔴	If a failure of the PL subsystem persists for [TBD] sec HK PL shall contact the EPS for a re	ContextSavingRequirement	RB	Edit	Delete

Figure 3.14: RERD's screen for Requirements Editing.

3.4 Evaluation case studies

3.4.1 CubETH case study

The CubETH nanosatellite [45] is comprised of: the electrical power subsystem (EPS), the command and data management subsystem (CDMS), the telecommunication subsystem (COM), the attitude determination and control subsystem (ADCS) and the payload (PL). Our early validation study is focused on the software for the following subcomponents of the CDMS subsystem (cf. A.2.1): 1) the CDMS status that resets internal and external watchdogs; 2) the Payload that is in charge of payload operations; 3) three Housekeeping components that recover engineering data from the EPS, PL and COM subsystems; 4) the CDMS Housekeeping which is internal to the CDMS; 5) the I2C_sat that implements the I^2C bus protocol; 6) the Flash memory management that implements a non-volatile flash memory and its write-read protocol; 7) the s3_5, s3_6, s15_1 and s15_2 services that activate or deactivate the housekeeping component actions; 8) the Error Logging that implements a shared RAM region. The case study comprises 38 requirements, from which 57 properties were derived. The complete BIP model can be found in A.2.5.

Table 3.12 summarizes statistics that characterize the utilization of BIP architectures. In total, the integrated architectures are 1 Action Flow, 11 Mode management, 5 Event monitoring, 10 Mutual Exclusion Management and 3 Failure Monitoring were used to enforce safety properties that have been derived from our boilerplates' requirements. Since safety properties enforced by each architecture are preserved by architecture composition (see Section 2.2), these safety properties are satisfied by the design model *by construction*. Table 3.13 shows the overal statistics that characterize the property enforcement step. In total, 38 requirements were formulated, from which 57 properties were derived. Among the derived properties, 4 were left to hold by assumption, 47 could be enforced and 10 were found to hold by inspection.

Table 3.12: Statistics on the utilization of BIP architectures

Model	**Flow**	**Mode**	**Event**	**Mutex**	**Failure**
Payload	0	2	0	4	0
HK PL	0	2	1	1	1
HK EPS	0	2	1	1	1
HK COM	0	2	1	1	1
HK CDMS	0	2	1	1	0
Flash Memory	0	1	0	1	0
CDMS status	1	0	0	0	0
Error Logging	0	0	1	1	0
Total	1	11	5	10	3

Table 3.13: Statistics of requirement formulation and property enforcement

Model	**Reqs.**	**Deriv. Prop.**	**Assum. Prop.**	**Enforced**	**By inspect.**
Payload	12	16	0	16	0
HK PL	4	6	0	6	0
HK EPS	4	6	0	6	0
HK COM	4	6	0	6	0
HK CDMS	3	4	0	4	0
Flash Memory	8	17	4	3	10
CDMS status	1	3	0	3	0
Error Logging	2	3	0	3	0
Total	38	61	4	47	10

Combined application of architectures can generate deadlocks. We verified the deadlock-freedom of the design model using the `D-Finder` tool [8]. `D-Finder`'s compositional analysis is sound, but incomplete: due to the employed over-approximation of reachable states, it can produce false positives, i.e., potential deadlock states that are in fact unreachable in the concrete system. However, our design model was found to be deadlock-free without any potential deadlocks. Thus, no additional reachability analysis was needed. The verification of deadlock-freedom was completed in 12 seconds, for our model consisting of 46 atomic components and 155 connectors.

The key advantage of our architecture-based approach is that the burden of verification is shifted from the final design to architectures, which are considerably smaller in size and can be reused. In particular, we managed to enforce 47 out of 57 derived properties using our simple architecture styles. The remaining 10 derived properties were verified by inspection and 4 fairness assumptions were left for verification using the `nuXmv` model checker.

Table 3.14 summarizes the duration of each process step for the input of the problem size shown in each row; the three roles of the process were performed by an engineer who was fully familiarized with the process's tool support. The property derivation and property enforcement steps are not shown, since they are fully automated and the time needed was negligible. We note that the time spent is not evenly distributed across the steps and it tends to be less towards the end of the process. Also, it is essential to clarify for the shown times that the architecture styles had already been configured in the RERD tool and the input forms for the style parameters had been defined. This takes 1–2 hours per style. Much greater effort was needed, though, to create the taxonomy of our architecture styles which took about 1 man-month. However, this taxonomy serves as a knowledge base in abstract form that we have acquired, and which can be reused to build other models of satellite on-board software.

Table 3.14: Durations and input sizes of the process steps

Step	**Duration**	**Input size**
Requirement specification	8 hours	38 requirements
Initial design	5 hours	12 components
Architecture instantiation	3 hours	47 enforced properties
Verification of deadlock freedom	12 seconds	46 components

3.4.2 Telecommand Management of an earth observation satellite

In a second case study, our model-based approach was also applied to an extract of 29 software requirements for the Telecommand Management function of a low orbit earth observation satellite. The requirements and the BIP model of this study cannot be disclosed, due to confidentiality liability terms. We derived 58 properties from the requirements and 34 (58%) of them were eventually enforced through architectures.

More specifically, during this case study we identified the need for and formulated an architecture style for Priority Management [28]. In overall, the integrated architectures were 10 Action Flows, 3 Mutual Exclusion Management, 13 Mode management and 1 Priority Management. The number of components in the BIP model was 25.

3.5 Related Work

The early validation of system requirements and its design using formal methods has attracted the interest of noteworthy industrial research initiatives [23], [46]. On the other hand, the principles of correctness-by-construction in system design have been introduced in [20] and [1]. In all technical approaches for correct-by-construction system design it is assumed that requirements and early design coevolve through iterative cycles [47], and the process converges into a design model, which (provably) fulfills all formal properties that are derived from the requirements. Existing works following the principles in [20] advocate a top-down hierarchical decomposition of the system into components. Correctness by construction is based on assume-guarantee contracts, where *assumptions* are either assertions on component inputs or invariants, and *guarantees* correspond to component requirements. Such top-down design flows [47]–[50] are concerned with the *allocation of system requirements* to system components (as in [51]), so that higher level requirements are established. System decomposition leads to the decomposition of contracts through a formal refinement relation [52]. When allocating requirements to a component, it should be ensured that the assumptions made for its environment (assertions or invariants) can be fulfilled. Developing assumptions manually is hard and the advantages when compared with monolithic verification have been questioned [53].

Our work aims at a bottom-up rigorous design flow [1]. Important differences from the top-down approaches are: (i) we focus on requirements formalization, rather than their allocation to components, (ii) we aim at the transformation of system requirements into a procedure, as opposed to the ad hoc design of components that should meet their contracts. Architectures in BIP drive the choice of system decomposition, component coordination and behavior transformation. In the top-down design flows, these choices should be validated through a posteriori verification; finding a solution in such approaches has a non-negligible complexity [53].

The use of natural language boilerplates in the formalization of requirements is not new. In [54], the authors target the specification and analysis of stakeholder requirements, referred to as *early requirements* [18]. Our approach for the use of boilerplates resembles those in [55], [56] and the CESAR reference technology platform [25]. CESAR introduces the Requirements Specification Language (RSL) that combines boilerplates of three clauses, namely the prefix, the main part and the suffix. Boilerplate attributes are defined in an attribute ontology and their placeholders must be filled with concepts from a *domain-specific ontology*. In [57], the authors introduce contracts with assumptions and guarantees built up from instances of RSL property patterns. A tool called DODT [58] allows for projectional requirement editing and for checking pairwise ontology-related contradictions [59] among requirements. Finally, properties are specified based on a recommendation of patterns with formal semantics, although no exact association of boilerplates with patterns is proposed.

The Easy Approach to Requirements Syntax (EARS) [26], [27] has introduced a set of structural rules (templates) for natural language requirements. The authors of EARS admit that their technique is mostly suitable for high-level stakeholder requirements and it is not applicable to all types of system requirements. Empirical evidence from industrial application showed improvement or, in some cases, complete elimination of problems related to ambiguity, vagueness, omissions and others. The EARS-CTRL tool [60] aims to ensure well-formedness in EARS requirements by construction and checks whether a controller can be synthesized from the provided set of requirements. If a controller cannot be synthesized, possibly conflicting requirements exist. The tool allows for projectional requirements' editing, based on a glossary defined on the domain of controller synthesis. Requirements are analyzed as LTL (Linear Temporal Logic) formulas. The analysis' effectiveness depends on user-defined semantic information (e.g. simple predicates) for the given glossary. Moreover, model synthesis is limited to a fragment of LTL that involves the universal path quantifier (**G**), the next-step operator (**X**) and the weak until temporal operator (**W**) [61]. Synthesis for such specifications is in PSPACE, whereas full LTL synthesis is intrinsically complex (2EXPTIME-complete).

Instead of automated model synthesis, we opt for incremental system construction that maintains the traceability of requirements up to the final design solution that discharges the derived properties. In this incremental process, designers can (re-)use "ready-made" solutions formally encoded in BIP architectures, which have been proven correct practically and theoretically. In essence, the architectures represent design patterns (e.g. for mutual exclusion, clock synchronization, scheduling, resource management, security) that are defined independently of the components which make up the system. We can thus ensure correctness-by-construction with respect to properties, while avoiding computationally expensive techniques that imply state explosion.

The importance of software architecture has been greatly acknowledged by the industry and academia. As a result, there has been an increasing interest in defining languages that support the architecture-based approach, e.g. UML [62] and architecture description languages (ADLs) [63], [64]. All these works rely on the distinction between behaviors of individual components and their coordination in the overall system organization. These languages, however, often lack formal semantics [62], [65], [66]. As a result, analysis is carried out on models that cannot be rigorously related to system development formalisms. This introduces gaps in the design process which reduce productivity and limit the ability for ensuring correctness. In fact, in a survey conducted in the industrial sector regarding architecture description languages, it is stated that practicing architects nowadays emphasize the need to reconcile informal notations with more formal and analysable ones [67].

Similarly to the aforementioned approaches, BIP architectures also provide a clear separation of concerns between functional and coordination aspects. BIP architectures have rigorous semantics; the underlying theory of components and their interactions is inspired from the BIP framework [2]. In essence, BIP architectures are operators restricting component behavior for enforcing a characteristic property. Their composition has some similarities with architecture composition in architecture languages with CSP-like semantics, e.g., Wright ADL [68]. Nevertheless, in contrast to these approaches application of BIP architectures does not require any modification of the components it is applied on. Additionally, as explained above, BIP architectures are tightly related with characteristic properties, which are preserved through composition.

3.6 Discussion

The applicability of our approach in an industrial context depends on a number of factors that we discuss henceforward. First, we assume the availability of a conceptual model like the one depicted in Figure 3.4. Such a model represents the structural elements and their conceptual constraints comprising the problem domain [69]; its adequacy and completeness determines the range of available concepts and relationships for the boilerplate attributes, the initial design, and the property derivation steps. We consider that conceptual modeling is performed by the Requirement Engineers in cooperation with the domain experts in charge of system design. This activity also includes capturing the *domain assumptions*, i.e., common and often tacit knowledge for the problem domain, and in spite of the system under design [30], [70]. The so-called domain knowledge (cf. Section 3.2) may concern with standardized protocols, services, libraries or physical laws, and can provide additional semantic information about the nature of the concepts in question. This information is essential, in order to conclude e.g. that certain events or data ranges that respect the conceptual model syntactically, are not relevant semantically. Some assumptions may be related to physics, e.g. “mass cannot be negative”, and some assumptions may be mission-specific, e.g. “the temperature within the orbiting range of the spacecraft cannot rise above N degrees”. Elicitation of domain knowledge, as a collaborative effort, could be facilitated by the use of templates for each ontology class [71].

The conceptual model and all assumptions related to domain knowledge are encoded into domain-specific and system-specific ontologies, which are accessed through the RERD tool. New concepts may be created from within the tool and the user is notified for violations of constraints related to the model integrity (e.g. undefined relationships). The model quality (syntactic, semantic, pragmatic) [39], [40] is a responsibility of the Requirement Engineers, who should aim for models that can be reused to significant extent in multiple projects. Certainly, the reusability depends on the *abstraction level of design*, since the requirements are usually specified at different abstraction levels along the development lifecycle (for space systems we have the spacecraft, avionics and software levels) and a conceptual model is pertinent only to a specific level [72]. The aforementioned problems and the right ontology in relation to our model-based design process need to be further researched in future work.

A second important issue is the expressiveness of the boilerplate language, and whether it can be sufficient for specifying the full range of requirement types found in the design of, say, space systems. This of course depends on the expressiveness of the property patterns, and on the analyzability of BIP models with extended semantics for the various property types, because correctness-by-construction does not vanish the need for a posteriori verification. The structure of the boilerplate language in Section 3.2.1 resembles that of RSL in the CESAR reference technology platform [25]. We currently support fewer templates than RSL for the prefix, main and suffix clauses, but this set of templates was sufficient for expressing the requirements of the case studies. Moreover, the RERD tool was designed such that new templates may be added; the only prerequisite is that the additional templates must be associated with property patterns, as in Table 3.7. The adopted framework of patterns from [43] is well-established and stems from industrially-relevant studies, but it only covers functional property specifications. We certainly foresee the need for boilerplates with templates for extra-functional aspects, which call for support by e.g. timing patterns [73] and probabilistic patterns [74]. It is worth to note here:

- the extension of BIP [75] that allows specifying probabilistic aspects of BIP components, while providing a stochastic semantics for the parallel composition of components through interactions and priorities;
- the RT-BIP extension for modeling timing constraints as a timed automaton, and a real-time engine for computing the schedules meeting the timing constraints, given the underlying platform's real-time clock [76].

These extensions are accompanied by advanced verification tools, some of which implement scalable compositional verification techniques [77].

However, a matter of vital importance is how expressive can a boilerplate language be with respect to today's industrial practice of natural language specifications. The loss of expressiveness, given the controlled vocabulary for the attributes, is inevitable though necessary to avoid ambiguity. However, the true question is whether it is still possible and whether we really need to cover all system aspects of today's specifications. This question also matters for languages like EARS [26], [27], which insist on natural language specifications using a fixed set of structural rules (though the EARS-CTRL analysis works with a user-defined glossary of terms). From our experience with the case studies we believe that only a subset of requirements needs to be validated. This subset includes requirements that may cause consistency issues and have to be established or checked against the system's structure and behavior.

Requirement Engineers tend to classify requirements in project documentation into categories (e.g. at the software level of space systems there are various classes of interface requirements, performance requirements, functional requirements and design/construction requirements). Any boilerplate language is considered adequate if it can express all representative forms of requirements that need to be validated, for all requirement categories (e.g. the design/construction requirements need not be expressed using boilerplates). This may imply changes to the scope of individual requirements (e.g. a natural language requirement may be broken into multiple boilerplate requirements). To this end, the RERD tool displays applicable boilerplates for each category of requirements found in a user-defined catalogue (Figure 3.14).

Our emphasis lies on precisely capturing the requirements by properties which—ideally—can be enforced through BIP architectures or—if not enforced—could be verified. As we aim to a semi-automated formalization of requirements, we are intentionally limited to specific types of requirements and templates. Our approach can accommodate additional templates for requirement boilerplates, provided that they are associated with property patterns, for which it is known how they can be enforced or verified.

The applicability of the correctness-by-construction approach throughout our model-based process depends on a library of BIP architecture styles for enforcing a worthwhile set of properties in the different categories of requirements. We have implicitly adopted the commonly accepted perception that the system's architectural design is in some sense intertwined with requirements [78], [79]. While specifying system requirements, Requirement Engineers have in mind the overall structure of the system under design (functional and physical architecture inputs shown in Figure 3.2) and a significant part of specification comes from adapting requirements found in previous projects. Our notion of architecture styles provides the means to formally capture common solutions to recurring design problems in an abstract and reusable form. This certainly incurs a non-negligible investment cost towards developing adequate and organized libraries of architecture styles, especially since the set of property patterns that they can enforce has to be precisely defined. The set of styles was derived by identifying commonalities in the base of natural language requirements of the case studies. Additional effort is required to this respect, whereas a recent research work opens prospects for defining styles that enforce quantitative properties [80].

Another important issue is the scalability and the effort needed for applying our model-based process. Indicative figures for problems of the size of our case studies have been previously mentioned. We acknowledge that in industrial problems of moderate size additional challenges may arise. More specifically, it may be trickier to identify and uniquely determine—on a team basis—the concepts for specifying requirements, as well as to verify properties against a large-scale design model. A-posteriori verification with model checking does not scale well and it can be rendered infeasible for large-scale models. With the architecture-based design a key advantage is, in particular, that the burden of verification is shifted from the final design to architectures, which can be reused. Moreover, as was illustrated in the case studies, the verification of deadlock freedom—which is essential when combining architectures—with the compositional approach of D-Finder is very fast.

However, when a non-enforceable property is not verified in Step 6 of our process, identifying a relevant sub-model for corrective action is complex. An important issue is how to present the resulting BIP model to engineers in a cognizable manner. In any case, the complexity of locating a design error is not inherent to the proposed process: it arises for any design process involving verification. In that sense, our proposal *improves* the current state of practice by reducing the number of properties that need to be verified.

Chapter 4

Compositional execution semantics for BPEL programs

4.1 Introduction

Businesses rely more and more on distributed, value-adding software applications in order to offer enterprise functionality to customers. Business Process Modeling (BPM) is a promising paradigm for integrating software components into a single executable unit, termed as process. The Service-Oriented Architecture (SOA) suits to the BPM paradigm, with respect to the composition of services into processes, which can be also deployed as services. Among existing languages for the specification of such processes, BPEL stands out by providing high-level primitives, and constructs for the definition of complex synchronous and asynchronous web service interactions. The used web services are autonomous and loosely-coupled components that possibly span different organizations. For the wide adoption of business process programming, it is essential to ensure reliability in order to avoid errors that may cause critical losses to the involved organizations. Additionally, the program has to fulfil correctness goals such as process responsiveness and compliance with partner services.

One approach towards ensuring reliability is by testing the process with emulating its interactions [81]. In this case, an adequate coverage of the program's control flow has to be achieved by selecting the appropriate test inputs. On the other hand, formal verification guarantees full coverage of execution paths for all possible inputs. Such an analysis has to be based on a formal specification of the language execution semantics, which involves nesting of service interactions using concurrency, isolation, compensation and event handling constructs.

Many works attempt to verify correctness by model checking a *formal model*, which is an abstract representation of the service composition program [82]. However, the original structure of the source program is not reflected in the formal model, thus rendering impossible to exactly locate the verification findings in the program's code. This is an inherent problem of most formalisms, which lack sufficiently expressive composition primitives for a model representation that preserves the service composition structure. BIP [2] provides a minimal set of primitives adequate for preserving the service composition structure. It consists of an executable modeling language for layered transition systems, which has formally defined operational semantics and mathematically proven expressiveness [83].

Therefore, we use BIP to introduce a compositional semantics for BPEL, i.e. a semantics in which the processing for each BPEL construct is placed locally to a corresponding BIP component. Such a definition tackles the combinatorial problem of defining semantics for each possible combination of nested BPEL constructs. Compositional semantics can be defined for executable languages with nesting syntax if the execution semantics of enclosing and nested constructs can be defined independently from each other. To achieve such a definition in our approach, the semantics of nesting constructs are defined based on abstractions built-in by construction for the nested ones, while the latter are combined using coordination primitives that do not alter their semantics (just restrict their execution traces). A structure-preserving translator into the BIP language has been implemented that covers all activities of the BPEL standard. The translator transforms BPEL programs into BIP models that contain the code needed for the verification of essential correctness properties. Properties are checked by exploration of the reachable state space. If a property is violated, we are able to obtain a counterexample execution trace that contains the processing steps of BPEL activities, which lead to the error location.

In [84], we presented a first version of our translator for a limited set of BPEL constructs with more emphasis on the translation algorithm. The verification of a functional property for a showcase application scenario was also demonstrated along with evidence for its violation in the form of a counterexample. Here, we expose:

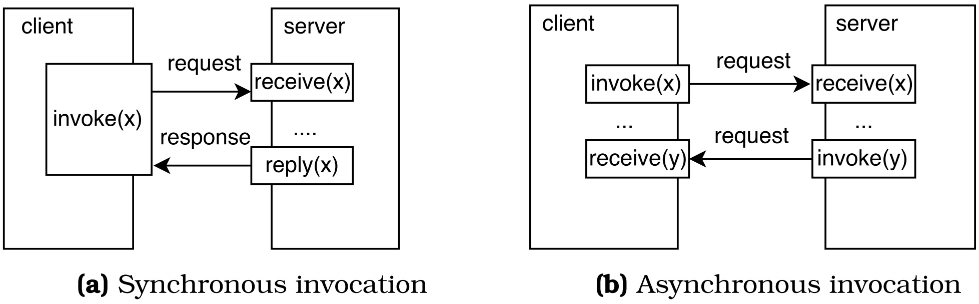
- the complete execution semantics of BPEL through a new methodology for compositional definition;
- the verification of a wide range of important correctness properties;
- the testing of our translator in mid-scale programs and their verification.

We note that the translation times were found to have a statistically significant *linear relation* to the number of states of the generated BIP model. The translator, the verification utilities for the properties of interest, as well as the BPEL programs of our experiments are available online in [85]. Verification is only one of the possible uses of our BPEL process models, which can be also used e.g. for test case generation based on the produced execution paths [86]. Moreover, in an independent research work [87], our approach was extended towards enabling the configuration of information flow policies for BPEL processes. Finally, our BPEL process models can be be executed as standalone web services, by being enhanced with runtime support for SOAP-based communication, through the application of the BIP architecture for SOAP-based web services that we presented in [88].

In Section 4.2, we discuss the design problems and the correctness of BPEL processes through a motivating example. Section 4.3 introduces the structure of our BIP model and the principles of the compositional approach for the definition of the BPEL execution semantics. These principles determine the interface and the behavior of BIP components, which allow implementing the semantics of the various BPEL activities. Section 4.4 encodes the BPEL execution semantics into safety properties that are *enforced in our model by construction*. Our modeling approach covers all activities of the BPEL standard, but the presentation is restricted to the most important activities and details for more activities are exposed in B.2. In Section 4.5, we present the verification of essential correctness properties that have been previously introduced in Section 4.2 and the formalization of additional useful correctness properties. Section 4.6 discusses the principles of the translation of BPEL programs in BIP. Section 4.7 shows results from the translation and analysis of mid-scale BPEL applications and the chapter concludes with a critical review of the related work in Section 4.8.

4.2 Correctness of BPEL processes: a motivating example

BPEL programs depend upon web services (partner links) whose interfaces expose *service operations* written in the WSDL 1.1 language. Synchronous operations accept an input and block the invoker for the output, or a fault, to be returned. On the contrary, in asynchronous operations the invoker dispatches the input and forgets it. Thus, through the use of two asynchronous operations it is possible to apply a request-response interaction pattern that does not block the invoker. In this approach, a service is invoked with the first operation and the response is returned with a second operation, referred to as *callback*, exposed by the invoker. The use of asynchronous operations generally allows for complex service interaction patterns, such as parallel operation invocations, but it raises the need to effectively manage communication *sessions*, i.e. the stateful chains of dual service interactions. The assignment of messages to the correct session takes place by message *correlation*.



(a) Synchronous invocation **(b)** Asynchronous invocation

Figure 4.1: Client and server side activities for synchronous and asynchronous invocations between processes.

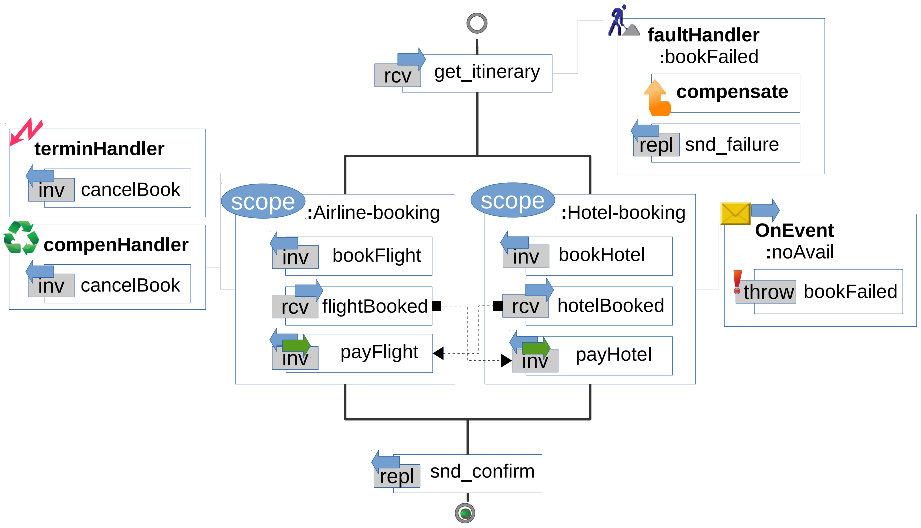


Figure 4.2: The *TravelBooking* process interacts with the *BookAirline* and *BookHotel* web services on behalf of a client.

Atomic behavior in processes is realized with *basic activities*, such as the `invoke`, `receive`, and `reply`, which are used respectively to (i) invoke, (ii) receive input, and (iii) send output (or fault), with respect to specific service operations. Figures 4.1a and 4.1b show the client-side and server-side activities used for a synchronous (resp. an asynchronous) invocation of an operation `x`. A client-side synchronous invocation is implemented by a request-response `invoke`, while the asynchronous interaction relies on an one-way `invoke` of `x` and a `receive` of the callback operation `y`. Generally, the `assign` activity is used before sending and after receiving a message, in order to copy data between the message and the process's variables. BPEL's *structured activities* define workflows of activities, such as `sequence`, parallel `flow`, and other conditional and repeatable structures. The `scope` activity defines a local context for its enclosed activities, with its own data and error handling through compensation, termination and fault handlers. A scope also defines event handlers for incoming messages and timeouts.

Example 9. *A BPEL process for travel booking is presented in Figure 4.2 with its activities shown in rectangular boxes. The activities for service interactions are labelled with the invoked operations. The bold, the thin and the dotted edges represent respectively relationships for the order of execution, the containment of handlers and the synchronization between activities.*

The process provides to its clients the synchronous operation get_itinerary that responds with an output or a fault message. When a client wants to book a travel itinerary, a get_itinerary request is received along with the preferred hotel, room type and flight details. Two scopes are then executed in parallel that communicate respectively with the HotelBookWS and AirlineBookWS web services:

- *The Hotel-booking scope invokes the asynchronous bookHotel operation of HotelBookWS to reserve the chosen hotel room. For this purpose, it uses an one-way* `invoke` *and continues its processing, while the response is pending. A* `receive` *waits for the confirmation in the hotelBooked callback operation. When the confirmation is received, the synchronous payHotel operation of the HotelBookWS is invoked for the payment. The progress of the whole process is then blocked on the synchronous* `invoke`*, until the receipt of the expected response. In parallel to the normal flow, the scope also has an event handler that listens to requests for the noAvail operation. This is a callback operation that is invoked by the HotelBookWS service, if there is no availability for the chosen hotel room. Upon receipt of such a message, the event handler throws a bookFailed fault.*

- *The Airline-booking scope invokes the asynchronous bookFlight operation of the AirlineBookWS to book the flight and upon the confirmation receipt the synchronous payFlight operation is invoked for the payment. The booking cancellation of AirlineBookWS is invoked, when the scope is abruptly terminated or when it is compensated after having been completed.*

Two synchronization links (dotted arrows) are used between the two scopes, in order to exclude the invocation of payment in each scope, before the confirmation is received in the other scope. The process responds to the client with a confirmation, after having completed the payment in both scopes.

Faults in any of the two scopes are propagated to the process level, where they are handled by the process's fault handler. In this case, the enclosed scopes are terminated, if they are still running, or compensated, if they have finished. Afterwards, the process replies to the client with a fault message. △

The main principles for the design of BPEL processes are summarized along the following three axes:

1. all available input from partner services must be received and handled
2. all the expected input to partner services must be provided
3. the process must hold a global view of the behavior that is composed

For the first axis, input from partner services is received though incoming invocations. A dedicated `receive` activity should be therefore *reachable* at each point of the control flow, where such an input is expected. In Example 9, the *hotelBooked* callback is invoked, when the *HotelBookingWS* responds with the result of the asynchronous *hotelBook*, and the *noAvail* callback is invoked upon a booking failure. If the input to *noAvail* was neglected, the process would be blocked forever in the `receive` of the *hotelBooked* callback, unless there would be a timeout handler to limit the waiting time. Moreover, with respect to the process's requirements, the input to *noAvail* is essential to throw a fault at the process root level in order to terminate the whole booking attempt.

For the second axis, for each partner service, the syntax of its expected input is defined in WSDL, but there is no standard way for specifying the input's semantics and its relation to other events. In Example 9, the *cancelBook* request to the *AirlineBookWS* is sent, to cancel a booking request. The termination and the compensation handlers of the scope must send this cancellation request, otherwise the process will have sent and saved booking results that are not reflected in its state. If the cancellation request is omitted, the *PayFlight* request will still be expected from the partner service, though it will not be sent. However, the relationship between an operation and its callbacks is not explicitly defined and the process's runtime environment cannot detect and handle the missing inbound or outbound responses of the asynchronous requests. Due to this weakness, the responsibility for handling the issue is delegated to the process designer.

For the third axis, when implementing a scope, it is important to consider the environment in which it is executed. In Example 9, if the *Airline-booking* scope was the only scope running within the process, there would not be need for implementing a termination handler, because there would be no other scope to throw a fault while this scope is executed. The termination handling is necessary, due to the parallel execution of the scope with a scope which might fail. If a termination handler can never be triggered, we have a case of *dead code* in the process.

The BPEL processes need to fulfil several *correctness properties* that are application-agnostic, in order to ensure safety of the control flow and the sessions. Some of these properties are BPEL constraints that are identified by *standard types of faults* (e.g. the *conflicting receive* fault). Essential properties that are not BPEL constraints, though they have to be ensured for any BPEL process, are the following:

1. *No blocking*: The process will not be blocked indefinitely for receiving an incoming message.
2. *No dead code*: The process does not include code that cannot be executed.
3. *Process termination*: The process can always terminate.
4. *No incomplete asynchronous request-response patterns*: The process cannot terminate with asynchronous (outgoing or incoming) request-response patterns that have not been responded.

The absence of dead code is important, in order to save memory space for the process execution and for the caching of operations by the CPU. A code segment may be unreachable, because of a logical error, such as conditions that are always false, or due to obsolete event handlers for messages that are not sent any more by the partner service. The process termination is essential for most processes that shouldn't run forever. A process may not be able to terminate, due to a possible livelock, i.e. an execution path of tasks that are executed infinitely often. The last property concerns the behavioral compliance between the process and the partner services with respect to asynchronous invocations. That is, if an invocation has been received (or sent) by the process, and this invocation is part of a request-response pattern, then a response will always have to be sent (resp. receive).

The main approach to address the aforementioned correctness and other application-specific issues is through testing [81] and interactive simulation. In an effective testing approach the developer has to create test cases that cover the program's control flow up to an acceptable level, as well as to instrument the program with assertions to be checked. The test case generation includes the definition of input/output data for the emulation of process interactions. When an assertion violation is encountered the developer has still to explore the execution path the leads to the violated assertion. This is possible through interactively simulating the BPEL process for the specific test case. Testing is an iterative procedure, since every time that the process is changed towards correcting an error, all test cases have to run again to ensure that no other assertion is violated. Moreover, due to the complex synchronous and asynchronous web service interactions, the cost for effectively testing a BPEL process may be much higher than that for testing other types of applications.

In this chapter, we introduce a modeling approach based on BIP along with the verification of the essential properties that is discussed in Section 4.5.1. Moreover, our approach can address additional verification needs that are detailed in Section 4.5.2. Within the development cycle of BPEL processes, correctness verification has to take place after process specification. As opposed to testing, verification is the only way to check functional properties against a process model with full coverage. If a property is violated, the needed corrections for the BPEL program are identified based on the automatically generated counterexample execution trace [84]. If all essential correctness properties have been verified, then extra-functional aspects can be addressed, such as the verification of security, e.g. through an extension of our approach in [87], or timing properties, e.g. through statistical model checking [89]. The verification procedure is iterated until all properties are met.

4.3 BIP model for BPEL processes

4.3.1 BIP components and model structure for the BPEL activities

Every BPEL process is defined by a single topmost scope that encloses variable declarations and stateful web service interactions through exchanged messages. Such a process is translated into BIP by preserving the structure of the used activities. Each activity is represented by a BIP component, called *activity component*, which explicitly defines the processing of that activity by the BPEL engine. *Basic activities*, like service invocations, are represented by the atomic BIP components of Table 4.1, apart from the `assign` activity, which is represented by one or more `copy` components.

BIP comp.	Behavior description
receive	handles a message receipt
reply	handles reply to a message
invoke	invokes a service operation
compensate	activates compensation of one or more scopes
valid	validates a variable's value w.r.t. its definition
empty	null behavior
exit	activates abrupt interruption of the process
throw	generates a fault
rethrow	re-throws a caught fault
copy	sets value to a variable, partner link or property

Table 4.1: Atomic components for basic activities of BPEL.

BIP comp.	Behavior description
data	manages access to the scope's data
timer	fires timer events
listn	handles multiple message receipts
links	manages access to synchronization links
rdlnk	reads a set of synchronization links
wrlnk	sets a synchronization link
loopctrl	controls loop execution
condctrl	controls conditional execution

Table 4.2: Atomic components for the BPEL semantics.

Structured activities are modelled as BIP compounds that enclose lower level activity components, as well as additional atomic components from those listed in Table 4.2. For every structured activity a set of connectors is generated, which is referred to as the *glue*, and may be accompanied by coordinating components that are attached to the glue. These coordinating components are introduced in Section 4.4 and they are not shown in the overall model structure.

`PROC` is the topmost component of the BIP model for a BPEL process. Its enclosed components specify the process's normal behavior (`norm`) and its associated fault-handlers (`faulthlrs`), that are executed as a response to a thrown fault. One more component is used to store the shared `data`, such as the process variables. Other shared data are the *partner links* and *correlation sets*, which identify services and communicating sessions respectively. The `norm` component encloses a primary activity (`act`) and event-handlers (`evhlrs`), that are scopes activated by timers or message receipts.

The overall model structure is further exemplified using phrase structure rules, where each rule refers to a compound (enclosing component shown in "<" and ">") in the left part and its constituents (enclosed components) in the right part. The high level structure discussed so far is reflected by the following two rules:

⟨*PROC*⟩ ::= ⟨*norm*⟩ ⟨*faulthlrs*⟩ data
⟨*norm*⟩ ::= ⟨*act*⟩ ⟨*evhlrs*⟩

The components shown in the right-hand side of the `PROC` rule are also used in `scope` components, along with two more handlers: the compensation-handler (`comphlr`) that specifies behavior for the reversal of the scope's effects, and the termination-handler (`termhlr`) that controls the forced termination of the scope.

⟨*scope*⟩ ::= ⟨*norm*⟩ ⟨*faulthlrs*⟩ ⟨*termhlr*⟩ ⟨*comphlr*⟩ data

The lower-level structure of the mentioned handlers is given by the following rules:

⟨*evhlrs*⟩ ::= ((listn | timer) ⟨*evscope*⟩)+ | empty
⟨*faulthlrs*⟩ ::= (catch ⟨*act*⟩)+
⟨*comphlr*⟩ ::= ⟨*act*⟩
⟨*termhlr*⟩ ::= ⟨*act*⟩

with the "|" representing the allowed alternatives and the "+" showing the occurrence of one or more components of the preceding type. The `evscope` activity component has the same structure with the `scope`, though it has a different glue. This allows to address the prescription of the BPEL standard, which foresees a special scope for event handlers that treats the handler's starting activity (modelled by a `listn` or `timer`) as if it were part of its main activity.

The execution of parallel activities can be synchronized through synchronization links: an activity can defer other activities through outgoing (source) links and can be deferred through incoming (target) links. The components `source` and `target` handle the associated links using one `wrlnk` and one or more `rdlnk`.

⟨*source*⟩ ::= rdlnk+ wrlnk

⟨*target*⟩ ::= wrlnk rdlnk+

The rules and the details for modelling each activity are as follows:

⟨*act*⟩ ::= ⟨*source*⟩? ⟨*target*⟩? (receive | reply | invoke | compensate | valid

| empty | exit | throw | rethrow

| ⟨*assign*⟩ | ⟨*seq*⟩ | ⟨*flow*⟩ | ⟨*loop*⟩ | ⟨*pick*⟩ | ⟨*if*⟩ | ⟨*scope*⟩)

⟨*assign*⟩ ::= copy+

⟨*seq*⟩ ::= ⟨*act*⟩+

⟨*flow*⟩ ::= links ⟨*act*⟩+

⟨*loop*⟩ ::= loopctrl ⟨*act*⟩+

⟨*pick*⟩ ::= ((timer | receive) ⟨*act*⟩)+

⟨*if*⟩ ::= condctrl ⟨*act*⟩+

where "?" specifies optional occurrence of the preceding component type. Structured activities enclose one or more activity components and control their execution either sequentially (`seq`, `if`, `loop`), in parallel (`flow`), or by deferred activation (`pick`).

4.3.2 Interface and behavior of activity components

The observable behavior of activity components [90] fulfills certain *assumptions (model abstractions)*. This enables the compound components to be defined based on the assumed observable behavior of the combined components i.e., *the glue is not tied to the combined behaviors.* For each activity component, its assumed observable behavior is ensured by construction, throughout the incremental building of the BIP model.

The fact that the glue is not tied to the combined behaviors allows the processing of each activity, i.e. the handling of control and dataflow events, to be *placed locally to the corresponding activity component*. The dataflow events comprise ports for reading/writing data to components that manage shared data. The control events include: (i) *commands*, that tell an activity what to do (e.g. start or terminate), and (ii) *notifications*, that are generated by the activity (e.g. upon finishing or throwing a fault). The *basic interface* of activity components consists of ports for the propagation of distinct commands and notifications.

Command ports of activity components are fired *by their outer* components, with the aim to:

- start them (*start*);
- disable them (*dsbl*), for some execution scenario;
- terminate them (*term*), due to a fault in a parent scope;
- compensate them (*rvs*), during the compensation of a whole scope.

On the other hand, notification ports are fired by activity components, when they:

- throw a fault (*fault*), which can be any BPEL- or WSDL-defined fault;
- have finished (*fin*), either successfully (not disabled/terminated or having thrown a fault) or unsuccessfully;
- complete their compensation (*rvsd*).

The observable behavior of an activity component with respect to some goal, is formed by hiding the actions that are of no interest for this goal. Thus, the component might have different observable behaviours for different goals. Let us consider the behavior of the `empty` activity component in Figure 4.3 and that of the activity component in Figure 4.4, whose τ and *fault* actions are hidden. We define the bisimulation relation $R = \{(e0, s0), (e1, s1), (e1, s3), (e1, s4), (e2, s2)\}$ between the states of the two behaviors, i.e. one relation such that the related states imitate each other's observable actions leading to states that again are related. We say that any two behaviors are *observationally equivalent*, if and only if there is a bisimulation relation R with $(e0, s0) \in R$, where $e0$ and $s0$ are their respective initial states. It can be easily shown that the two behaviors in Figures 4.3 and 4.4 are *branching bisimilar* [91], which is an observational equivalence notion that preserves the branching structure of behaviors. This means that R preserves the computations *together with the potentials in all intermediate states* that are passed through, even if hidden actions are involved.

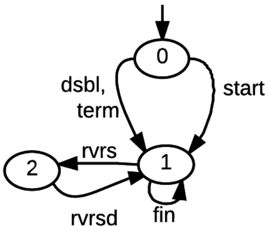


Figure 4.3: `empty` activity behavior.

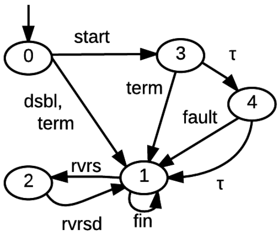


Figure 4.4: Example of activity behavior.

One general assumption for defining the glue is that the behavior of all activity components is branching bisimilar with the behavior of the `empty` component. This assumption is ensured by construction for all activity components; in atomic components, this is easily implemented in their behavior, which consists of limited control locations, while in compounds, the glue and possibly additional coordinating components enforce the externally observable behavior.

The state of service interactions

Since the lifetime of service interactions may span the execution of multiple activity components, the `data` components have to accommodate variables for sharing the service interactions' state. These variables store: (i) the (url) location of partner link services, that may change dynamically, (ii) sets of *correlation properties* that are instantiated and accessed by activity components for service interactions (`receive`, `reply`, `invoke`, `listen`), (iii) the enabled Inbound Message Activities (IMAs) for routing messages to the listening `receive` components, and (iv) the open IMAs, that identify incomplete inbound synchronous requests. More details for the representation of the service interactions' state and the detection of associated faults are given in B.1.

BPEL variables

The variables of a BPEL process store the messages' content or other business specific information that has to be shared among the activities (of a scope or globally), which may influence the control flow. Their data types are either XML types or WSDL message types with partitions, called *parts*. In our model, XML typed variables and variable parts are represented as local variables in the `data` components, which are accessed by activity components with dataflow processing, such as the `copy` and `receive` components. The `read` and `write` ports of activity components are used to read and assign variables of `data`, respectively.

For the values of BIP variables , we have adopted a data abstraction approach (details are given in B.1), which allows to identify (i) variables that have not been initialized, (ii) pairs of variables that hold the same value, and (iii) variables that are not assigned within a loop body.

4.3.3 Atomic BIP components

Each atomic BIP component is parameterized according to the modelled activity. We adopt the common implementation approach of BPEL engines, which serialize the execution of basic activities. For this purpose, we enforce a mutual exclusion management in which the components for basic activities perform their execution (i.e., critical section) one by one; they remain blocked until they get the lock through their *allow* port, and invoke their *done* port to release the lock.

As an example, we show in Figure 4.5 the `receive` component for a synchronous operation and we discuss its differences from a `receive` component for an asynchronous operation. The `receive` is completed in two consecutive processing phases, one for the establishment of message listening, and one for the processing of the received message. The actions in these phases are included in two critical sections and the component remains idle in-between. Specifically, the component includes ports in order to:

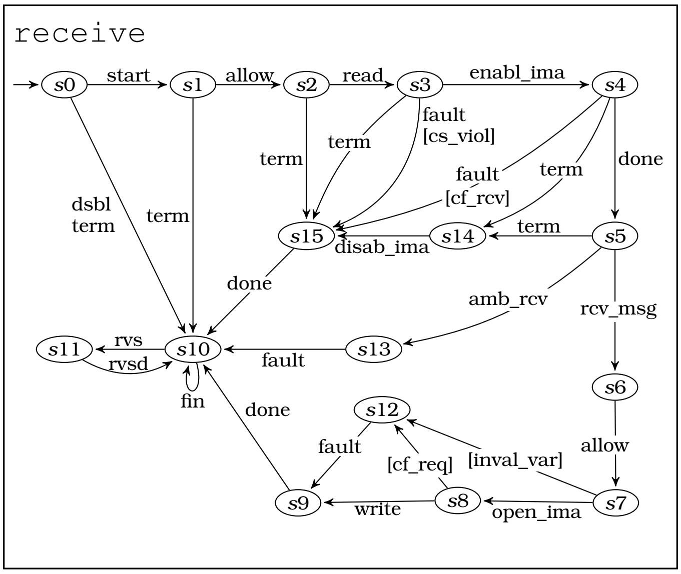


Figure 4.5: `receive` component for synchronous operation.

- read the expected partner link and correlation sets (*read*),
- establish message listening (*enabl_ima*),
- stop message listening (*disab_ima*),
- detect an ambiguous receipt (*amb_rcv*),
- receive the message (*rcv_msg*),
- *allow*, to get the execution rights in order to begin processing
- enlist the request identifier (*open_ima*),
- store the message and the correlation sets (*write*).
- *done*: yields execution

Component actions are guarded by the boolean variables shown in brackets, which represent detected faults. Possible faults are the *correlation violation* (cs_viol), *conflicting receive* (cf_rcv) and *conflicting request* (cf_req). Moreover, the *ambiguous receive* fault is detected when the *amb_rcv* port interacts with another listening component and the *invalid variable* (inval_var) is thrown if the message does not match the expected structure. , and a WSDL-defined fault (fault_msg) if the message is a WSDL fault message.

The `receive` component for asynchronous operations does not have the *open_req* port, since there is no need to enlist the requests of asynchronous operations; thus, no conflicting request faults are thrown in this case. In B.3, we expose the details of the other atomic components.

4.4 Compositional semantics definition

The execution semantics is enforced onto the composed behaviors of BIP compounds through coordination defined by the glue and additional components. A compositional definition is enabled by the principle that the coordination is not tied to the combined behaviors (shown in Section 4.3.2) and preserves the execution semantics of combined behaviors (no additional behavior is introduced, as explained below).

The acceptable behavior for the BIP compounds representing structured activities is captured in the form of safety properties defined over ports. To ease readability, all properties are defined using natural language statements. A set of *general safety properties* for any compound $\mathtt{C}$ with n enclosed components $\mathtt{A}_1 \ldots \mathtt{A}_n$ is the following:

- if $\mathtt{C}$ is disabled (*dsbl* port), so do all $\mathtt{A}_i$.
- if $\mathtt{C}$ is terminated (*term* port), so do all the $\mathtt{A}_i$ that can be terminated.
- • $\mathtt{A}_i$ can atomically cause the component to throw a fault, if the component does not handle it.
- $\mathtt{C}$ is finished (*fin* port) only if all $\mathtt{A}_i$ are finished.
- • $\mathtt{C}$ is not finished before disabling each $\mathtt{A}_i$ that will not start.
- $\mathtt{C}$'s compensation is completed (*rvsd* port) only if all $\mathtt{A}_i$ have been compensated.

Moreover, the order of compensation (*rvs* port) for all $\mathtt{A}_i$ is the reverse order of their *start* port activation, if there is an imposed order (e.g. for sequence). Otherwise, the compensation of all $\mathtt{A}_i$ is started simultaneously.

The aforementioned safety properties, as well as the invariants for basic activities and additional safety properties specific to each structured activity aim to formally capture the informally defined BPEL semantics from [92]. The invariants for the basic BPEL activities are enforced within atomic BIP components by design, such as in Figure 4.5 and in the components of B.3. For the structured BPEL activities, we introduce architecture styles that enforce safety properties - like those mentioned - associated with their semantics. This means that each property is built-in by construction within the used architecture style, i.e. it is implied by the behavior of the coordinating component(s) plus the used glue.

In the BIP compound for a structured activity, the architecture styles are instantiated into concrete architectures by defining a mapping from the styles' parameters to the compound's enclosed components (operands). This involves also a mapping of the parameters' ports to operands' ports. According to the results presented in [14], the safety properties of combined architectures are preserved in the compound; this result is also valid when the architectures are composed hierarchically. Moreover, all compounds only interfere with the lower level components by applying synchronization on their ports and synchronization always preserves the component invariants. In this way, we follow the principle of compositional semantics definition mentioned in the first paragraph of this section.

The following properties and the used architecture styles are specific to the most important structured activities. Details for the other activities are exposed in B.2.

4.4.1 BIP compound for the `flow`

Definition 4.4.1. A `flow` compound encloses one `links` component and n components $\mathtt{act}_1 \ldots \mathtt{act}_n$ that contain k `rdlnk` and m `wrlnk` components in total. The following properties have to be satisfied:

- if `flow` is started, so do all $\mathtt{act}_i$.
- some $\mathtt{rdlnk}_i$ can read a link (from `links`), only if some $\mathtt{wrlnk}_j$ has set the link (to `links`).

For the properties of the `flow` compound, two architecture styles were combined, namely the *Parallel* style in Figure 4.6 and the *Synch. links mngmt* style shown in Figure 4.7. The *Parallel* style enforces the first property of Def. 4.4.1 and the general safety properties that hold for any compound. The *Synch. links mngmt* enforces the second property of Def. 4.4.1.

All architecture styles are applicable to operands that: (i) have at least the ports assumed for the replaced parameter, and (ii) are branching bisimilar with the behavior assumed for the replaced parameter. For example, for the parameters $\mathtt{A}_i$ of the *Parallel* style (Figure 4.6) we assume the ports of the basic interface and the behavior of the `empty` component. The style is applicable to $\mathtt{act}_1 \ldots \mathtt{act}_n$ that fulfill these assumptions for $\mathtt{A}_i$. The style's coordinator `P` mediates the interactions of the basic interface of all $\mathtt{A}_i$ with the environment, so that the coordination fulfills the general assumption of Section 4.3.2: the observable behavior of the compound is branching bisimilar with the `empty` component. If the compound is started (`P`.*start*), all $\mathtt{A}_i$ are started due to a rendezvous connector ($n : 1$ means that the connector connects all $\mathtt{A}_i$). Therefore, the first property of Def. 4.4.1 is enforced, when the style is used in the `flow` compound. Furthermore, the connectors in Figure 4.6 that connect the *dsbl*, *term* and *fin* ports enforce the general safety properties.

The *Synch. links mngmt* style manages the access of m parameters `WRlnk` and n `RDlnk` to the synchronization links of the `Lnk` parameter. Each `WRlnk` sets a link through the *set* port, whereas each `RDlnk` reads a set of links through the *get* port. Ports *set* and *get* are connected to `Lnk` through the WRVAR (resp. RDVAR) connectors that enable exchange of data. A `RDlnk` can read a set of links (`RDlnk`.*get*), only if some `WRlnk`s have set these links (`WRlnk`.*set*), since each `Lnk`.*get* port is assumed to be guarded with this condition. Therefore, the second property of Def. 4.4.1 is enforced, when the style is used in the `flow` compound. Since for the `RDlnk` and `WRlnk` parameters we assume a trivial behavior with a single state, the style is applicable to the `rdlnk` and `wrlnk` components of `act` (*set* and *get* ports are exported by `act`).

4.4.2 BIP compound for the `scope` and `PROC`

Definition 4.4.2. A `scope` compound encloses the components (i) `norm`, (ii) `faulthlrs`, (iii) `termhlr`, (iv) `comphlr` and (v) `data`. Let us also consider that the enclosed components (i) to (iv) contain k `rddat` and m `wrdat` components

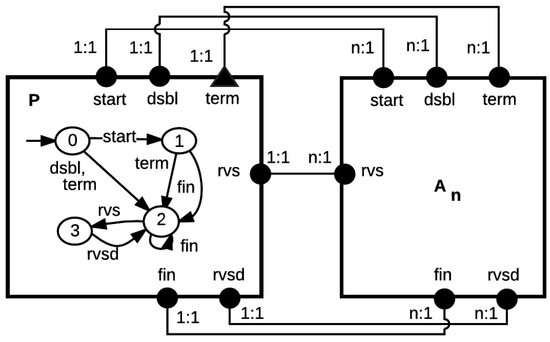


Figure 4.6: *Parallel* style

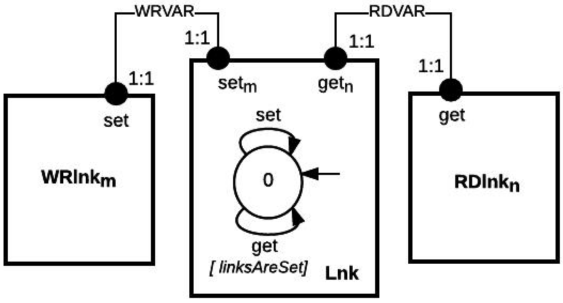


Figure 4.7: *Synch. links mngmt* style

in total, which read (resp. assign) variables stored in data. The following properties have to be to satisfied:

1. if scope is started, then norm is started.
2. if norm throws a fault and norm is terminated while termhlr, comphlr are disabled, then faulthlrs is started only if norm has been terminated.
3. if scope is terminated while norm is executed, norm is terminated and also faulthlrs, comphlr are disabled, then termhlr is started only if norm has been terminated.
4. scope is finished successfully, only if norm has finished, without being disabled, terminated or having thrown a fault.
5. scope is finished unsuccessfully, only if faulthlrs or termhlr have been executed and finished.
6. if faulthlrs throws a fault, the fault is thrown by scope, then faulthlrs is terminated only if scope is terminated (i.e., by the enclosing scope that will handle the fault).
7. comphlr is started, only if a successfully finished scope is compensated.
8. scope is finished only if norm, faulthlrs and termhlr are finished and while comphlr is not being executed.
9. the values of variables in data are modified only if they are assigned by some wrdat.

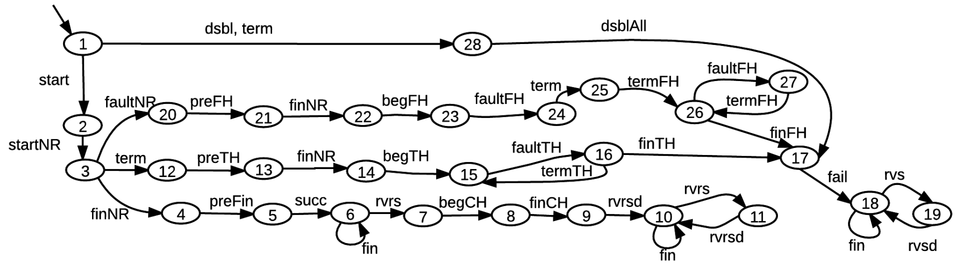


Figure 4.8: Coordinator of the *Scope* style

For the properties of Def. 4.4.2 two architecture styles were combined, namely the *Scope* and one *Data mngmt*. The *Scope* style is used only in scope, in order to fulfill properties (1) to (8). The style has a coordinator that is shown in Figure 4.8 and takes as parameters the norm (NR), faulthlrs (FH), termhlr (TH) and comphlr (CH). The coordinator's role is to export the scope's basic interface and to coordinate the parameters with respect to their basic interface. This happens under the assumption that faulthlrs, termhlr and comphlr are branching bisimilar with the empty. The coordinator enforces property (1) by starting norm (*startNR* port) after the scope is started (*start* port). For property (2), a fault thrown by norm (*faultNR* port) is followed by the preparation needed for starting fault handling (*preFH* port), that involves terminating norm and disabling faulthlrs and comphlr. Afterwards, norm must first finish (*finNR* port) before faulthlrs is started. Property (3) is treated accordingly. For property (4), the coordinator enables the *succ* port after norm has finished without being terminated or thrown a fault. Similarly, for property (5), the coordinator enables the *fail* port after either faulthlrs (*finFH*) or termhlr (*finTH*) is finished. For property (6), the coordinator enables *fault* after faulthlrs has thrown a fault (*faultFH*). Then, it waits for the scope's termination (by the enclosing scope that will handle the fault) before invoking the termination of faulthlrs (*termFH*). Property (7) holds because the coordinator disables comphlr whenever the scope is not going to finish successfully, during the preparation for termination (*preTH* port) and fault handling (*preFH* port). Finally, property (8) is enforced by starting comphlr (*startCH*) after scope's compensation is invoked, provided that scope has finished successfully.

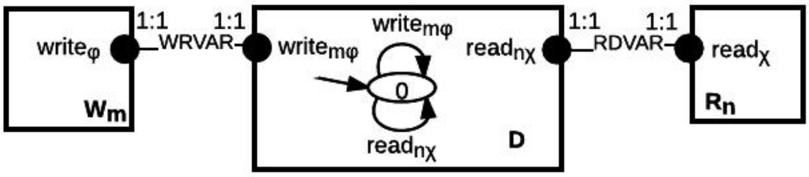


Figure 4.9: The *Data mngmt* style

The *Data mngmt* style in Figure 4.9 is used to fulfill property (9). The style has one parameter D that stores some variables, m parameters W, and n parameters R. The W have φ ports for assigning variables, whereas the R have χ ports for reading variables. The style is applied to each scope by mapping the wrdat and rddat components to the W and R parameters, while the data component is mapped to D. Property (9) is fulfilled by connecting data exclusively with the wrdat components that are enclosed by the scope.

The PROC compound uses the *Proc* style, whose coordinator is different from the coordinator of the Scope style with respect to the following four aspects: (i) it does not have the branches starting with *term* and *disbl*, since PROC is a root component and cannot receive events that come from enclosing components, (ii) the *faultFH* is not followed by the faulthrs termination, but immediate interruption of the process occurs instead, (iii) if an exit component is executed within norm or faulthlrs (i.e. invoking coordinator's *exit* port), then the interruption of the process occurs, and (iv) it does not need to enable the ports *succ*, *fail*, *rvs* and *rvsd*.

4.5 Verification of correctness properties

In the verification procedure of Figure 4.10, we attach a fault injection component, as well as observer automata (monitors) [37] for observing the state of the BPEL process model, whereas the state space exploration takes place with one of the BIP tools [34]. The building of monitors, that is discussed later, requires information given by the user in a configuration file. The procedure steps are shown with rectangles and the input and output data for each step with dotted lines. More specifically, the individual steps are as follows:

Input: (i) the BPEL program and its WSDL definition files , (ii) the configuration file

Output: the verification verdict

Step 1 *BPEL-to-BIP translation.* The BIP model is built together with the monitors and the fault injection component through the translation of the BPEL program and its WSDL definition.

Step 2 *Verification.* The verification output is produced by executing the BIP state space exploration tool. The output contains diagnostic messages appended by the monitors, the activity components and the BIP exploration tool.

Step 3 *Analysis of verification.* The verification verdict for each property is produced. For some properties, it suffices to inspect the diagnostics in the verification output, while for others a post-processing program is used[1]. If a property is found to be violated, a counterexample execution trace is generated through post-processing the verification output with a dedicated program.

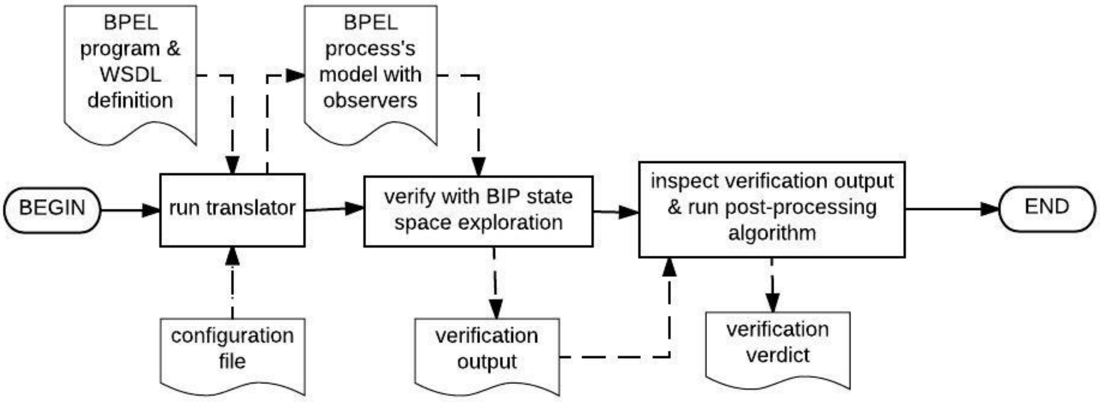

BPEL program & WSDL definition
BPEL process's model with observers
BEGIN
run translator
verify with BIP state space exploration
inspect verification output & run post-processing algorithm
END
configuration file
verification output
verification verdict

Figure 4.10: The BPEL process verification procedure.

In the following subsection, we discuss specifically the verification of the essential correctness properties from Section 4.2. Moreover, the verification of other important properties (standard BPEL faults, properties specific to the application functionality and compliance with a session specifications) is discussed in Section 4.5.2.

[1] The post-processing program and the configuration file template are available online in [85].

4.5.1 Essential properties

No blocking

To detect blocking by *receive* activities, we compose the process model with a fault injection component that consists of two non-deterministically selected states: the state representing availability of partner links for communication (AVAIL state) and the state in which partner links cannot send messages (NAVAIL state). This component also exports a *send* port, enabled at the AVAIL state, which is synchronized using a rendezvous connector with the *rcv_msg* ports exported by the `receive` components. If the process model can be blocked by an incomplete `receive`, then the state-space exploration will detect deadlocks in the execution paths, in which the NAVAIL state was reached. Indeed, the *rcv_msg* ports will not be triggered if the NAVAIL state is reached and a deadlock will be met unless each `receive` is stopped by a `timer` component. The attachment of the fault injection component only affects the model by restricting the execution traces to those observed when a failure in the process environment exists. Thus, the execution semantics of the process is not changed.

Process termination

We detect livelocks caused by eternal loops that prevent process termination. In particular, we detect execution paths in which the variables used in the loop exit conditions are not updated during the loop's execution. For each evaluation of an exit condition, the values of variables are compared with the values they had in previous evaluation. This check is integrated within the `loopctrl` components, which print a diagnostic message during state exploration when reaching states where the aforementioned condition is detected.

No dead code

Dead code consists of basic activities, which are not started in at least one execution path. Such activities are detected using a script that processes the output of state-space exploration. During the exploration, each basic activity component prints its id twice: in one message at the initial state and in a second message once it is started. The output is post-processed by a script that adds the ids of the messages printed at the initial state in a hash set and removes the ids of the messages at starting. The ids that remain in the hash set after the output is processed indicate the components that are dead code. Note that the post-processing algorithm scales linearly with the number of printed messages, since the cost for adding and retrieving each id in the hash set is O(1) in the average case.

No incomplete asynchronous request-response

In order to verify that every asynchronous request-response pattern is responded in all execution traces, we introduced two monitoring components for observing the incoming (resp. outgoing) patterns. The monitor of outgoing patterns observes the dispatch of the first message (request) to partner services and the receipt of the second message (response). Upon process termination, if the dispatched requests are more than the received responses, the monitor outputs the property's violation. Similarly, the monitor of incoming patterns prints a message, if more receipts of requests are observed than the dispatched responses. The attachment of monitors does not affect the execution traces, because they are connected with the activity components using broadcast connectors, through which only the activity components can trigger the monitors and not the other way around. Thus, the execution semantics of the process is not changed.

For this check, it is necessary to define which request and response messages are associated, since this cannot be derived from the WSDL description. This mapping is given by the user in a configuration file.

4.5.2 Additional correctness poperties

Our BPEL process models can be used for verifying properties that are defined in the context of the application's functionality. For example, in [84], we verified a purchase order BPEL process with respect to the property: "If an invoice has been issued, the process must not complete before sending the invoice to the client". Such properties are verified through the use of application-specific observer automata.

The properties related to standard BPEL faults, such as those checked in [93] (message delivery atomicity, no session ambiguity, possible inputs) and [94] (the so-called conflicting receive in BPEL) are checked by the activity components, which throw these faults. Thus, we represent the handling of every such fault by the responsible scope.

Besides that, our model can be checked for compliance with the acceptable message sequences within a session, as it is demonstrated in [95]. This entails the attachment of an automaton representing the language of acceptable message sequences, while observing the exchanged messages identified by the session. The process's compliance is then verified, if the observer automaton is in an accepting state, when the process is terminated. Properties like those in (4) of Section 4.2 are examples of compliance with a language of sequences with two exchanged messages (asynchronous request and response). An example language for a complete session between the process of Section 4.2 and the Airline Booking service is shown in the observer automaton of Figure4.11. Transitions are labelled with send and receive message actions of the BPEL process that are allowed at each state. Valid message sequences lead to the states 4 and 5 which enable the port *valid*. If a message is exchanged that is not allowed, a non-accepting state is reached that is not shown and the message sequence is rejected. Thus, the process is not compliant with the session language specification, if there is an execution path in which it sends, for example, cancelBook twice.

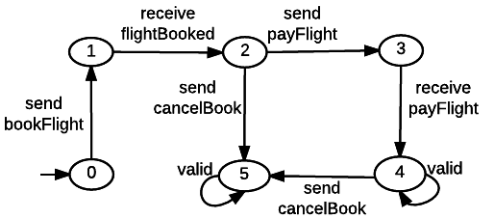


Figure 4.11: Observer automaton for a language of acceptable messages.

4.6 BPEL to BIP translation

A code generation embedding [96] was developed, such that BPEL is translated into BIP by parsing BPEL programs and their referenced WSDL descriptions.

Our algorithm implements a single pass syntax-directed translation [97] through a recursive decent parsing of the BPEL XML tree and a postorder call of the BIP code generation function for the tree nodes. The code generation function is invoked only for the XML elements corresponding to BPEL activities. A code fragment, that is generated upon each function call, is part of an incrementally built BIP model. The symbol table structure stores information derived from the parsing of the BPEL XML tree (e.g. visible BPEL variables) and the referenced WSDL descriptions, as well as from the code generation results. The latter is necessary for building a hierarchical model, in which each BIP compound refers to the enclosed BIP components and their exported ports.

The code generation function uses templates of code for BIP components with placeholders. The tokens of XML elements (tree nodes), such as the element tag and attribute values, determine the template to be used. The placeholders are replaced by BIP code that is generated based on: (i) information retrieved from the symbol table, and (ii) the attribute values of the XML element.

Figure 4.12 shows the template of a BIP code fragment that models the `copy` activity. The template starts with the declaration of the `copy` atomic component, which includes data (lines 3-4), a set of states (line 6) and transitions (lines 8-13). The template accepts two input parameters: () an auto-incremented component identifier i, () a list $var[k]$ of size K with component variables for storing the message parts. The placeholder for i is noted with <i>. The lines ending with /* for k=1..K */ comments (i.e. lines 4 and 10) are repeated for each element in

```
atomic type copy_<i>()
  /* ... sample of data ... */
  data int err
  data int <var[k]>   /* for k = 1 .. K */
  /* ... sample of states ... */
  place s0, s1, s2, s3
  /* ... sample of transitions ... */
  initial to s0
  on read_<i>from s1 to s2 do {
    if <var[k]>==-1 then err=10  fi    /* for k = 1 .. K */
  }
  on fault from s2 to s3 provided (err>0 )
  on write from s2 to s3 provided (err==0 )
end
```

Figure 4.12: Template of BIP code for the copy activity

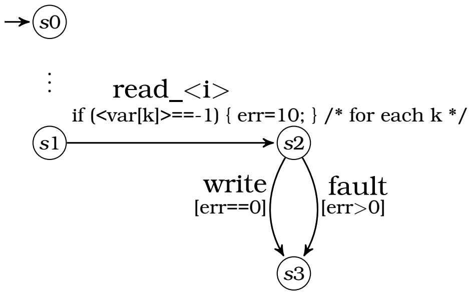


Figure 4.13: Behavior of the copy template

var[*k*]. The behavior corresponding to the transitions of this template is shown in Figure 4.13. Specifically, the copy starts from state *s*0. Let us consider that it reaches *s*1, where it reads the message parts that should be copied (port *read_<i>*). After checking whether there are uninitialized message parts, either the message parts are copied to the new variables (port *write_<i>*) or a fault is thrown (port *fault*).

The template of BIP code for the assign activity component is shown in Figure 4.14 and the component's structure is illustrated in Figure 4.15. The template starts with the declaration of the compound (line 1) and the enclosed copy components (line 3). Lines 5-7 show the declaration of a connector that synchronizes all the copy.*write* ports. This connector exports a *write* port at the assign's interface (line 9). A number of *read* ports are also exported, one for each included copy.*read* port (line 10). On the other hand, a single *fault* port is exported for all included copy.*fault* ports (lines 11-13). The template accepts three input parameters: () an auto-incremented component identifier

```
compound type assign_<i>()
   /* ... enclosed components ... */
   component <cp[k]> C<k>   /* for k = 1 .. K */
   /* ... sample of connectors ... */
   connector <wrConn> write_<i>1(
     C<k>.write , /* for k = 1 ..K */
   )
   /* ... sample of exported ports ... */
   export port write_<i>1.xpr as write_<i>
   export       port       C<k>.read_<cp[k]> as
read_<cp[k]>   /* for k=1..K */
   export port
          C<k>.fault,   /* for k=1..K */
   as fault
end
```

Figure 4.14: Template of BIP code for the `assign` activity

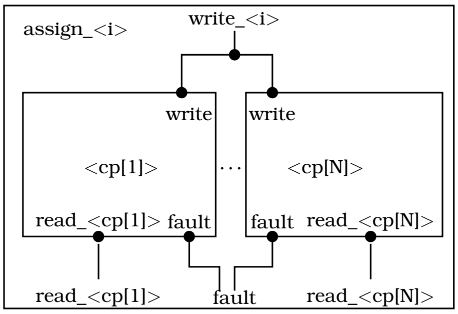


Figure 4.15: Behavior of the *assign* template

i, () a list $cp[k]$ of size K with the enclosed `copy` components () the connector *wrConn* used for the assignment of message parts.

4.7 Experiments on the verification of BPEL programs

Table 4.3: Statistics and verification results for analyzed BPEL processes.

Process ID	**# comp.**	**# conn.**	**# RSS**	**transl. time**	**verif. time**	**no block.**	**process termin.**	**no dead**	**no incompl.**
AmericanAirlines_12	42	216	12533	6471	21112	-	-	0 of 13	1 of 2
AmericanAirline_59	22	127	69	3632	8	-	-	0 of 4	1 of 1
BookRating_50	20	129	69	4496	18	-	-	0 of 4	0 of 0
BookStore1_52	40	243	1791	5118	1204	-	-	1 of 8	0 of 3
BookStore2_49	42	247	1791	4834	1211	-	-	1 of 8	0 of 3
BuyBook_48	85	454	2437	6441	567	fail	-	2 of 13	0 of 1
BuyBook_51	89	1359	49295	15967	43737	fail	-	29 of 54	0 of 3
BuyBook_53	50	836	13607	9933	3242	fail	-	2 of 25	0 of 5
BuyBook_54	27	326	983	6167	245	fail	-	0 of 15	0 of 5
BuyBook_55	34	341	1707	9796	349	fail	-	0 of 19	0 of 5
DeltaAirline_56	13	90	69	3736	15	-	-	0 of 4	0 of 1
Employee_57	12	89	69	3712	18	-	-	0 of 4	0 of 0
Travel_58	25	243	907	5721	209	fail	-	0 of 13	0 of 3
TravelApproval_41	168	811	32907	11089	24364	-	pass	0 of 27	0 of 0

The scalability of our analysis in real-size BPEL programs and the effectiveness of the verification approach were tested using a number of BPEL programs of various sizes from [98] and [99]. The programs mentioned in the rows of

Table 4.3 were first translated and then analyzed with respect to the essential properties of Section 4.5.1.

For each program, Table 4.3 summarizes in the corresponding row the statistics for the size of the BIP model, the translation/verification time and the obtained results. The shown times in *ms* were measured on a 64-bit machine with an Intel(R) Core(TM) i7-3770K CPU @ 3.50GHz and 16 GB RAM running Ubuntu 14.04. The first five columns show the number of components, connectors and reachable states (RSS), as well as the CPU time for translating and verifying the program. The worst time taken for the program translation was 16 sec, whereas for the program verification was almost 44 sec. The translation times were found to have a statistically significant *linear relation* to the number of states of the generated BIP model (B.4).

The last four columns show the verification results, for each one of the properties of Section 4.5.1. We note with dash the cases in which a property does not raise some correctness issue or a property's definition is not applicable. For example, we don't need to verify "no blocking" in programs, which do not wait for incoming messages other than the first messages that create new process instances. Also, "process termination" is not relevant to programs that do not have loop activities. The verification result for the two aforementioned properties is noted with either "pass" or "fail". For the two other properties, Table 4.3 shows the number of violation cases out of the total number of checked cases. For example, for the property "no dead code", it is shown the number of non-reachable basic activities together with the total number of basic activities. The last column shows the number of incomplete asynchronous requests and the total number of asynchronous requests.

The "no blocking" property is violated in all programs in which it was checked. By inspection of the execution traces it was found that only the programs BuyBook_48 and BuyBook_51 could avoid the eternal waiting for messages using timers, but not in all cases. The "process termination" property was relevant only for the TravelApproval_41 program, where it was found to be satisfied. The property "no dead code" was violated in five programs. More specifically, we detected fault handlers that cannot be started. Some of these handlers were included, in order to be invoked, if a partner service is not available. However, such a fault can be thrown only from BPEL engines, which support this non-standard fault. On the other hand, we do not provide built-in support for non-standard faults and for this reason the corresponding fault handlers cannot be started. Finally, two programs were found, in which the property “no incomplete requests” was violated. By inspection, we confirmed afterwards that these programs do not respond in all cases to services that invoke the process.

4.8 Related Work

In order to place our approach in the broad range of formal methods for the analysis of service compositions, we start with the comparison framework in [82]. In that article, the authors review the pros and cons of 35 related works classified in three categories of semantic models, namely automata or labelled transition systems, Petri-nets and process algebras. The authors conclude that few of the considered formal methods address in a satisfactory way the correctness properties for continuity of service delivery, and they specifically emphasize that only a limited number of proposals support exception handling and compensations, as is the case in our approach.

The works of [100] and [94] are the only BPEL formalization approaches accompanied by available translators (the BPEL2oWFN and the BPEL2PNML tools respectively). A comparison between the two frameworks is presented in [101]. The resulting models of both approaches can be checked by tools that verify temporal properties on Petri-nets. The models in [94] do not represent data dependencies but they can be checked efficiently by tools that analyze the structure of WF-nets. Such examples are the WofBPEL tool in [94], which can check application-agnostic properties and the tool in [102] that checks the conformance of a BPEL process model with respect to message logs. On the other hand, models in [100] are built with an abstraction level that suits the intended analysis goal, with the possibility to represent data dependencies.

These models can be checked for general and application-specific properties only by reachability analysis, since they do not comply to the structural requirements of WF-nets. Compared to the two aforementioned translation approaches, our work exposes the way that the translation rules ensure the BPEL semantics and provides more expressive means (i.e. the observer automata vs temporal logics) for the definition of application-specific properties.

The distinct feature of our approach is a compositional definition of execution semantics for BPEL, such that there is no need to model all possible combinations of nested BPEL constructs. Similar approaches are those in [103], for the BPEL, and in [104], for an artificial variant of OWL-S. The latter, does not feature complex event handling structures like the ones offered by BPEL. Moreover, the main goal of semantics definition was the implementation of an orchestration execution engine, as opposed to ours, which is the verification of correctness properties. In [103], the authors describe a stepwise refinement approach for a structure-preserving modeling of BPEL in Event-B. This work, like ours, is supported by a translation tool and considers both generic and custom correctness properties for verification. We instead follow a constructive approach in building the model by gradually imposing constraints while preserving invariants. In another work [105], a BPEL translation to FIACRE [106] is presented. FIACRE is a formal language for modeling both the behavioral and timing aspects of systems. This work is focused on the translation approach, rather than a compositional semantics definition. No details are provided for the verification of BPEL programs and for the scalability of their translation to programs of various sizes.

In [107], the authors suggest the use of process algebras for the specification and reasoning of BPEL processes. Their specification framework is based on CCS (Calculus of Communicating Systems), which allows only to focus on web service interactions, rather than also dealing with data exchanges and the complex event handler structures in BPEL. For considering the effectiveness of BIP in comparison with this and other process algebraic approaches, we recall [108], where the authors realized a semantic gap between BPEL and the π-calculus. They recognize that the notion of global state over BPEL computations, the message passing and the combination of sequencing with concurrency create interleaving and name binding behavior that cannot be faithfully represented in π–calculus. As a consequence of these findings, in order to provide formal semantics for the BPEL activities, they extend π–calculus with a transactional construct. In [109], a two–way mapping is introduced between BPEL 4WS and the LOTOS process algebra. The authors claim that LOTOS has the expressive power to structurally represent BPEL processes, due to its compositionality semantics. However, LOTOS lacks expressive primitives like for example the broadcast connector in BIP.

The difference between the process algebra setting and that of BIP is thoroughly studied in [83]. The components in BIP are characterized by their behavior (labeled transitions) and their composition takes place by means of interaction models and priority models, which essentially perform memoryless coordination of behavior. On the other hand, in process algebras processes evolve with the use of operators for composition. With respect to a notion of expressiveness that characterizes the ability of some framework to coordinate components, process algebras have been shown to be less expressive than BIP. Regarding the compositional semantics definition, the coordination means of process algebras do not preserve the invariants of composed behaviors, since the behaviors evolve.

Moreover, it is worth comparing BIP with high-level modeling languages for component-based systems. One language that has been used for modeling compositional construction of web services is Reo [110]. However, Reo's semantic model, which is the constrained automata, cannot preserve the BPEL process structure due to lack of powerful coordination operators, such as those in BIP. The lack of structure preservation, along with the fact that connectors in Reo are stateful, would complicate the definition of a compositional semantics definition for BPEL in Reo.

Finally, in an independent research work [87], our structural BPEL process representation from [84] was extended towards the verification of service non-interference, which is an interesting application of our compositional semantics framework. To do so, the process designer must provide a configuration file (.xml) with authorization rights, that is, a list of owners and authorized readers - partner link services - for critical data. Then, the configuration synthesis algorithm takes as input the BIP model of the BPEL program and the configuration file and builds information flow dependency graphs by considering all implicit and explicit data dependencies in the system. In case a total configuration file can be generated by the tool, the system information flow is then considered non-interferent with respect to the initially defined configuration. Otherwise, the system is interferent and the system designer has to re-define the initial configuration by utilizing the obtained counterexample. The calculated configuration is optimal, i.e. only data that need to be protected is configured as critical. The ultimate aim is to reduce the security processing overhead like cryptography encryption and decryption, signature calculation, certificate verification, etc.

Chapter 5

Application modeling for rigorous design of IoT systems

5.1 Introduction

The Internet Of Things (IoT) aims at the seamless interconnection of heterogeneous embedded systems using the Internet technologies and infrastructure. The connected *things* are network nodes equipped with low-memory and low-power devices that collect data from the surroundings (sensors) and communicate it to the system, as well as smart objects that perform computations. System integration is facilitated by operating systems [4], [111], [112], which enable the nodes' control by abstracting the provided hardware and software resources.

Many different programming models have been proposed for the various types of IoT applications [113]–[116]. In practice, software design relies heavily on node programming close to the operating system level. This affects the development time, the application reliability and its efficiency, due to the heterogeneity of the involved things and the diverse interaction modes that should be taken into account. In applications with a continuous communication, a fixed schedule with periodic transmissions has to be designed. On the other hand, if communication is event-driven, i.e., using dynamic scheduling, it may cause packet collisions and message overloading. Moreover, applications are not easily adapted to evolving needs, especially when they are deployed in large-scale distributed environments [117].

The design complexity of an IoT application is reduced significantly, when reusing web services [118], [119]. According to the REpresentational State Transfer (REST) style, things are accessed as abstract resources located by Universal Resource Identifiers (URIs) and they are manipulated through the Hypertext Transfer Protocol (HTTP) or the Constrained Application Protocol (CoAP) [120]. However, IoT applications primarily involve asynchronous interactions, as opposed to the synchronous interactions found in most Internet applications. Furthermore, a typical web service is a long-running executable process, whereas IoT applications are targeted for resource-constrained things, which may have to remain idle for relatively long periods of time.

To address the risks due to the overall design complexity, operating systems provide tools [121], [122] for simulation-based validation of execution scenarios. However, this analysis does not support an early validation of requirements at design time. To this end, a *model-based design process* relying on formal modeling semantics can open prospects for an exhaustive analysis of the application's behavior, and a simulation analysis grounded on statistical confidence.

We therefore advocate a design flow based on BIP. The system design is specified in a Domain Specific Language, which is used to preserve the consistency between the auto-generated BIP model and the application code. BIP is suitable for building executable models of mixed software/hardware systems. With its expressive coordination primitives, it facilitates the modeling of heterogeneous computations and interactions (synchronous and asynchronous) that are inherent in service-based IoT applications [1], [88]. It also allows using resource variables that model resources (e.g. time, memory, energy). Formal verification of the BIP model through state space exploration guarantees behavioural correctness properties. If the model is extended with stochastic variables, it is amenable to statistical model checking [123], a simulation-based analysis with statistical guarantees, i.e. finely controlled quality of results by various confidence parameters. This allows validating quantitative properties derived from requirements relevant to the IoT system's architecture [124].

Through analysis by state space exploration we ensure deadlock freedom and other properties related to handling events in a timely manner, i.e. what we call service responsiveness. With statistical model checking, we validate properties for the correct operation of the system, under statistical assumptions for its external stimuli, as well as the message buffer utilisation, the collision occurrence and the event queue blocking times [125]. These analyses are not supported by today's IoT operating system simulators. The concrete contributions of this research work are as follows:

- We present our model-based design flow in the context of IoT WPAN (Wireless Personal Area Network) systems.
- The domain specific language (DSL) for REST applications running on the Contiki OS [4] is then introduced; the focus on the Contiki OS was motivated by the fact that it is open source and its design is transparent to the development community.
- We illustrate the design of a REST sense-compute-control (SCC) application for a building automation system. The BIP model was based on the WPAN architecture standards and it was subsequently calibrated with respect to Contiki OS runtime constraints.
- We provide results for properties derived from requirements for IoT WPAN systems. Moreover, a study of the system robustness with respect to error behaviors is presented.

The described approach supports a component-based design philosophy that allows locating design errors to specific components, while boosting modular design and reuse of model artifacts. Through the separation of concerns, it enables the application development independently from the IoT system architecture. Finally, based on the wide range of supported analyses, multiple requirement types can be validated. The BIP models for the Contiki OS were first presented in [126]. Compared to that work, we introduce here the design flow with the DSL, and we present a SCC use case (a sense-only application was shown in [126]). This allowed us to additionally provide statistical model checking results for the system's operation under assumptions for its external stimuli.

Section 5.2 provides the background on the foundations of IoT systems. Section 5.3 introduces the model-based design flow. Section 5.4 presents the case study and Section 5.5 comments on the benefits and the limitations of our approach. A comparison with other design methods is also included.

5.2 Background

5.2.1 Foundations of IoT Systems

In an IoT system, the interactions among the different abstraction layers of the mixed SW/HW architecture (Figure 5.1) impact the overall system's performance and efficiency. From another point of view, we often have different applications overlapping in networks of heterogeneous things. The REST design is an architectural style that enables application-layer interoperability, i.e. multiple applications can co-exist and share the same set of things. It is used in the design of node-decentralized systems, but the nodes may be also integrated in cloud-centric solutions [127], which are more often preferred. Here, we only consider the former case. In most IoT operating systems, there is integrated support for the design of REST applications.

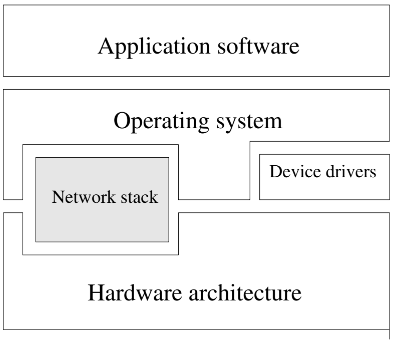


Figure 5.1: Typical SW/HW architecture of IoT node

Message exchanges involve a "server" and a "client". Clients send requests to servers and receive responses; they are responsible for keeping the state of the session with each server. Servers accept requests in the form of CoAP/HTTP methods (e.g GET, POST, PUT, DELETE), for retrieving or modifying the state of their *resources*; every resource is uniquely identified by a URI and encapsulates data, such as its description, location and state (e.g., sensed values, mode and other). The same networked node may act as a server in some communications (e.g. to provide access to sensor values), and as a client in others (e.g. to register in a directory). There are also intermediary nodes, which implement both the client and server roles in order to forward requests or translate them in other protocols. For instance, a gateway node is used to mediate client requests and forward them to the actual server; such nodes often encapsulate legacy services, to improve server performance through caching and to enable load balancing across multiple machines.

The IoT applications for WPAN systems can be classified into two broad categories according to how they manage their internal interactions and the interactions with the physical environment. The *Sense-Only* (SO) applications collect sensor data on occasions (intermittent sensing) or regularly. Such examples are the smart heating systems, where one can remotely access and adjust the in-house temperature using cloud services, as in [126]. On the other hand, applications of the *Sense-Compute-Control* (SCC) category, both sense and coordinate various control activities. Such activities range from e.g. taking proactive actions for controlling the temperature. The main difference between these two application categories is the autonomic operation of the SCC nodes, i.e. their functioning without any human intervention.

5.2.2 Contiki and REST application programming

The Contiki OS implements a lightweight architecture for *event-driven* applications [4]. The node processes wait for events and handle them in *event handlers* that run sequentially with respect to other handlers without being interrupted, i.e. they finish upon running to completion. This means that lengthy event handlers can absorb the processing capacity [128].

Processes communicate by posting *synchronous* and *asynchronous* events to each other. Synchronous events are dispatched immediately when they are posted, thus they are handled on the spot. The execution control returns to the calling process after the event handler has finished. On the other hand, asynchronous events are en-queued by the OS for later dispatch. While synchronous events target a single process, asynchronous events can target many processes, which are called in a sequential iteration.

Events are identified by their type. Along with the predefined event types by the OS, processes may define custom event types. Two commonly used predefined events are the *poll* and *exit* events. *Poll* is a high priority asynchronous event, which are dispatched before any other asynchronous event. *Exit* is posted synchronously and causes its target process to run the exit handler and eventually end. As opposed to other handlers, the optional poll and exit handlers are enabled at all control flow locations of the process.

The Contiki OS features a REST Engine API [129], for the definition of REST service resources. A basic REST resource has a name, a URI-path and a set of supported HTTP/CoAP methods. Each access is handled by a dedicated *resource handler* that handles requests coming from the network. A REST Engine process listens to the network stack and transfers the requests/responses from/to the resource handlers. The design of REST applications typically consists of the following steps:

1. The REST resource definitions are provided and the resource handlers are implemented or reused.
2. The application behavior is distributed to nodes. The server processes and the client processes are implemented.
3. Appropriate parameters for the network are configured.
4. The functional behavior is debugged using the OS's simulator.
5. When enough testing confidence is achieved, the client and REST server implementations are deployed on system nodes.

The Cooja simulator [122] is used to validate the functional behavior and, when simulating realistic workloads, the performance aspects of Contiki OS applications. However, such a simulation provides insight only for a limited set of execution scenarios and cannot ensure by itself the diverse requirements of an IoT system. Programming for resource-constrained devices involves various sources of delay that are hardy predictable. First, asynchronous communication within a node or between remote nodes is prone to delays, due to the execution of other processes and the fact that events are handled sequentially. Second, delays might occur due to message encoding and decoding, which depends on the overall CPU load of nodes. Finally, there is a high probability of packet collision in the network, if nodes access the communication medium simultaneously. In this case, all the involved nodes will back off the transmission for a random period of time and retry after this period has elapsed. With Cooja, the programmer is restricted to inspecting the behavior of application functions and the performance of system nodes without being able to inspect the interactions at the lower layers of the node architecture shown in Figure 5.1.

5.3 The BIP model-based design flow for IoT systems

An IoT project always starts from a set of requirements that can be classified in two major categories: (i) the functional requirements that are related to the application functionality, and (ii) the non-functional requirements related to the performance and the efficiency of the IoT system, along with its robustness characteristics. Functional requirements can be captured by formally specified safety and liveness properties, but we may be interested also for a quantitative characterisation of the correct operation of system under statistical assumptions for its external stimuli. Of particular importance are the non-functional requirements, such as the enforcement of bounded latencies due to e.g. packet collisions, and limited energy consumption.

We advocate a design flow using models that capture both the functional, as well as the non-functional aspects of behaviour across all abstraction layers of Figure 5.1, while supporting the separation of concerns during the system's design [130]. The separation of concerns is two-fold and involves not only the separation of application from the lower abstraction layers, but also the separation of computation from the communication. The former is of vital importance as it enables the application development independently from the IoT system architecture. The latter refers to the mechanisms and the primitives of the protocols employed in the network stack, which can be handled independently from the data processing. In this context, developers can model and build artifacts separately for the software and the various node architecture layers, which can be also reused in similar applications. Those reusable model artifacts are easily instantiated and parameterized, with respect to the particular system under design.

The design flow aims to the progressive development of the system, starting from the modelling and the implementation of the application functions up to their deployment onto the IoT system. The design philosophy is incremental in nature, since it is based on the hierarchical composition of simpler model artifacts, i.e. components, to form more complex *components*. An immediate consequence is that the debugging and identification of design errors in simpler components is easier and less time-consuming. In the course of the design flow, the developers should be able to find the optimal deployment - from the performance perspective at a given system scale - for the applications, while ensuring their proper functioning.

Our models rely on the BIP component framework. We take care of preserving the consistency between the BIP models and the corresponding application code by generating both from a single design definition written in a proprietary DSL. The DSL refers specifically to REST applications to be deployed onto IoT WPAN systems with Contiki OS nodes. The language provides XML-based constructs for the communication, the control flow and the scheduling of events in Contiki OS nodes, as well as for mapping the application's modules onto the system's nodes. The network configuration parameters are defined in XML files of the format proposed in [131]. Each model modification for fulfilling a violated requirement is respectively applied by the developer to the DSL design definition, from which the updated Contiki code is generated.

The design flow is based on analysis techniques for guaranteeing the qualitative and quantitative properties that capture the functional and non-functional

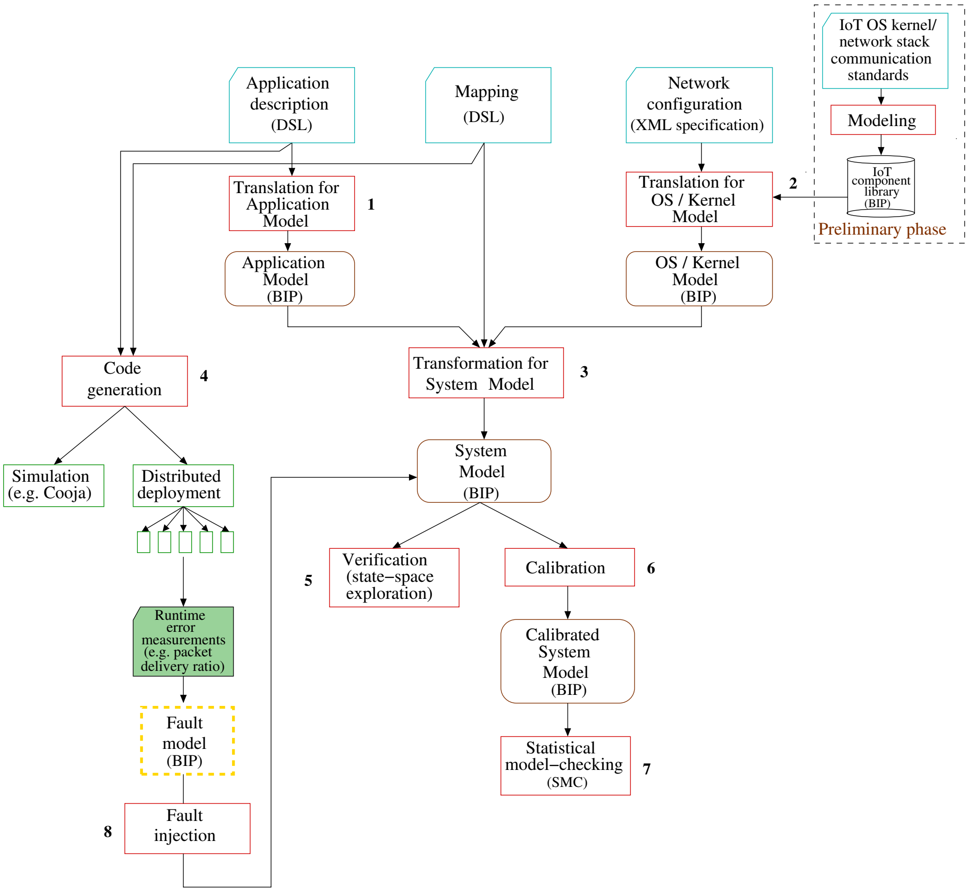

IoT OS kernel/ network stack communication standards
Modeling
IoT component library (BIP)
Preliminary phase
Application description (DSL)
Mapping (DSL)
Network configuration (XML specification)
Translation for Application Model
1
Translation for OS / Kernel Model
2
Application Model (BIP)
OS / Kernel Model (BIP)
Code generation
4
Transformation for System Model
3
Simulation (e.g. Cooja)
Distributed deployment
System Model (BIP)
Verification (state–space exploration)
5
Calibration
6
Runtime error measurements (e.g. packet delivery ratio)
Calibrated System Model (BIP)
Fault model (BIP)
Statistical model–checking (SMC)
7
8
Fault injection

Figure 5.2: The BIP model-based design flow for IoT WPAN systems (numbered rectagles show the process steps)

requirements. The qualitative properties for safety and bounded liveness are formalised as observer automata [37] for monitoring the state of the BIP model and are then verified with the BIP tools for state space exploration. The quantitative properties are validated with statistical model checking [123].

Figure 5.2 provides an overview of our model-based design flow. We assume the availability of a library with model fragments for the applicable OS and SW/HW network stack, which can be reused in new IoT projects. All the necessary system components are instantiated from this library, according to the DSL design definition, which includes the IoT application's mapping onto the system's nodes. From these artifacts, it is possible to generate the BIP model of the IoT system. The overall approach consists of the following steps:

1. **Translation for the construction of the Application Model.** The design definition for a REST application in the DSL is translated into BIP. The structure of the DSL description is preserved and this allows to trace the

analysis findings back to the design definition.

2. **Translation for the synthesis of the OS/kernel Model**. The BIP model fragments for the OS and the network stack (part of the network stack may be implemented by the HW architecture, as shown in Figure 5.1) are instantiated from the OS kernel library, through the translation of the XML-based network configuration file that determines also the components' parameterization and their interconnection.
3. **Transformation for the construction of the System Model**. The DSL mapping onto the system's nodes is manually edited[1]. It is used for transforming the Application and OS/Kernel models into a System Model. Appropriate BIP glueing code is generated from the defined mapping to connect the OS kernel model with the application model.
4. **Code generation.** The design and mapping definitions in DSL are used for generating code. The code can be either node-specific, i.e. ready to be deployed onto the distributed IoT system, or node-agnostic, if there is need for further validation within the OS simulator (Cooja). In any case, the code is deployed/simulated according to the specified application mapping onto the system's nodes.
5. **State-space exploration.** The verification of qualitative properties derived from functional requirements takes place by state-space exploration with the BIP tools. In case of a property violation, the original DSL definition (and the auto-generated BIP model) has to be repaired.
6. **Calibration.** The runtime characterization of the IoT application takes place by executing the generated code on the nodes and within the system environment. All influential hardware/ software constraints are identified and subsequently added to the BIP System Model.
7. **Statistical model checking.** The validation of quantitative properties derived from functional and non-functional requirements takes place by SMC. If a property is not satisfied at the required level of confidence, the DSL definition has to be repaired.
8. **Fault injection.** The robustness of the BIP System Model is analyzed

[1] The computation of the optimal solution would have to be based on a parametric mapping definition

by considering various error behaviours, such as loss of bandwidth and radio interference as additive noise. The former increases packet losses and out-of-order deliveries and the latter causes error-prone access to the wireless medium and increases packet collisions [132]. For this analysis, additional components with the error behaviour are added, and further parameterization through proper runtime error measurements is required [126].

The design flow is iterated upon any change in the application's description or in its mapping, which follows the previous verification and validation step. The Step 8 takes place only when there are specific robustness requirements for the system under design.

In any case, the step 4 precedes the verification and validation of requirements, in order to enable the calibration of the system's model in step 6. This process aims to augment the model with timing information for computations and communications from executions of the actual system. The timing information for computations is derived from executing an application process on a system node. Communication times are induced by functions that use communication resources. In general, these times depend on the system's interactions with the environment, and on various interference factors that are not known in advance. In these cases, we can apply a probability distribution fitting technique, whereas the functional BIP model is transformed into a stochastic BIP model through the introduction of probabilistic variables that represent stochastic time evolution [123]. When the software runs to the completion of events and if it is possible to justify the absence of any interference, the measured execution times can be approximated by fixed times that are incorporated into the BIP System model.

5.4 Case study

5.4.1 General description

The design of a building automation SCC application with digital and analog sensors is now illustrated using the BIP model-based design flow of Figure 5.2. The application controls the temperature and detects motion in offices using passive infrared (PIR) sensors. A ZIG001 Temperature-Humidity sensor is installed in each office, along with the low power MS-320LP PIR, both from Zolertia[2]. A zone-controller (acting as Client) reads temperature measurements (shown as t in Figure 5.3) and turns on a thermostat, if the temperature exceeds the desired level ($t > ub \vee t < lb$). During non-office hours, the controller runs into the energy-saving mode, and the desired temperature is reduced. If there is motion detected (Figure 5.4) during non-office hours, an intrusion alarm is activated. The intrusion alarm notifies the subscribed devices inside or outside the building (e.g. smartphones). If there is motion during office-hours, the lights switch on.

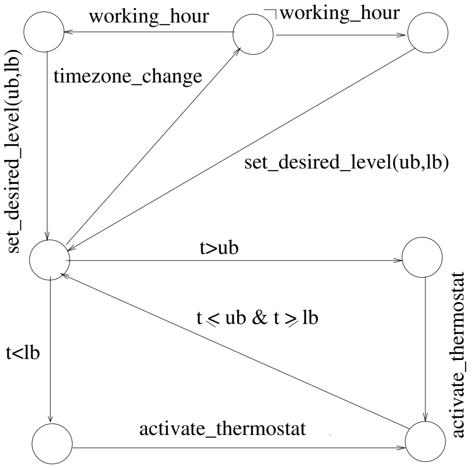


Figure 5.3: State flow for temperature control by the zone-controller (temperature t should be in $[lb, ub]$)

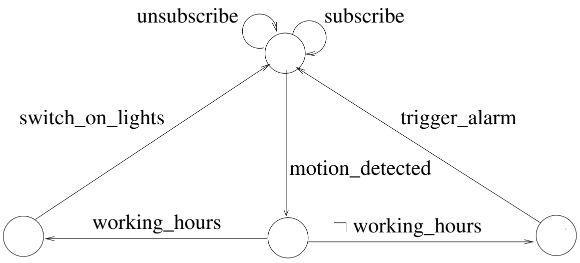


Figure 5.4: State flow for motion detection by the zone-controller (alarm triggered in non-office hours)

The system consists of 5 nodes with the zone-controller acting as client of 4 REST servers using CoAP (Figure 5.5). Each server owns a temperature resource for a ZIG001 and a motion resource for a MS-320LP PIR. The client runs two

[2] http://wiki.zolertia.com/wiki/index.php/Z1_Sensors

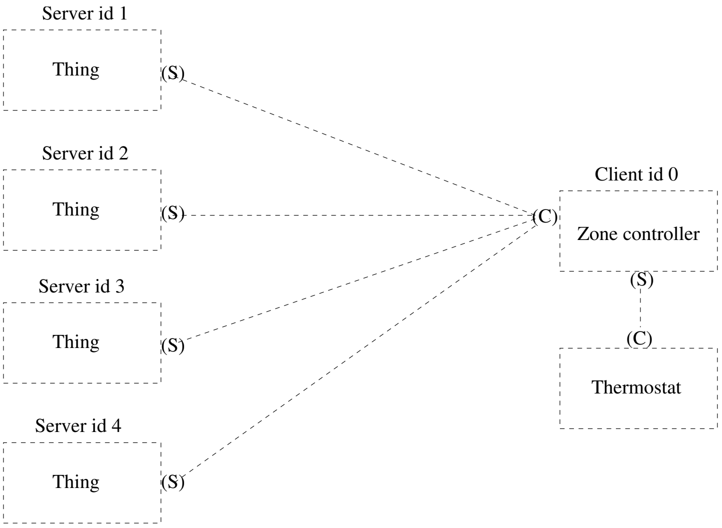


Figure 5.5: Node topology with clients and servers in the building automation system

Contiki processes, one for sending unicast GET requests to the servers and one for periodically observing their motion resource. The GET requests are sent with a fixed transmission period of 1s. The observation of a resource begins with a registration request and is cancelled with a de-registration request. Upon the state change of a resource, the server sends a CoAP notification to the registered client, which acts according to the time of day. Message receipts should be acknowledged by the server. However, if the client does not receive a response within a deadline, the request is re-transmitted.

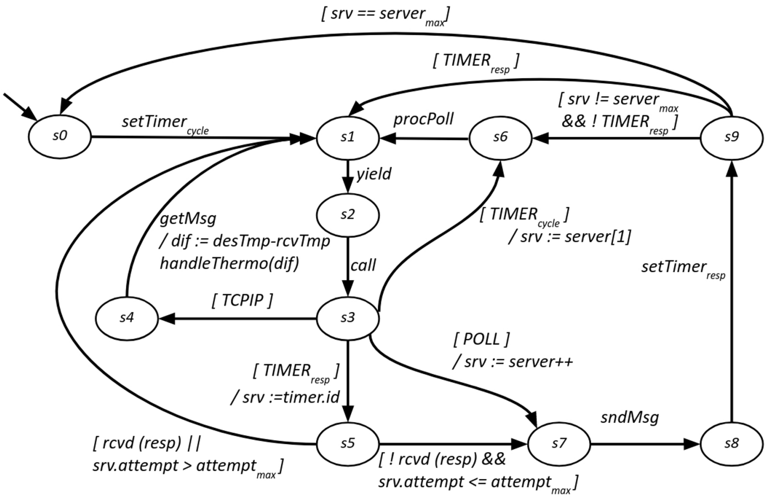

[srv == server$_{max}$]
[TIMER$_{resp}$]
[srv != server$_{max}$ && ! TIMER$_{resp}$]
s0
setTimer$_{cycle}$
s1
procPoll
s6
s9
yield
s2
getMsg / dif := desTmp-rcvTmp handleThermo(dif)
call
[TIMER$_{cycle}$] / srv := server[1]
setTimer$_{resp}$
s4
[TCPIP]
s3
[POLL] / srv := server++
[TIMER$_{resp}$] / srv :=timer.id
sndMsg
[rcvd (resp) || srv.attempt > attempt$_{max}$]
s5
s7
s8
[! rcvd (resp) && srv.attempt <= attempt$_{max}$]

Figure 5.6: Client process of the building automation application

Figure 5.6 depicts the behaviour of the client process's body for receiving temperature measurements[3]. A cycle of measurements starts upon a $TIMER_{cycle}$ event (timer expires). The process then sends a request (unicast message) to a server, and asks to be polled for contacting another server, until all servers have been contacted. For each request, a deadline timer is set ($setTimer_{resp}$) for waiting the response. If the deadline expires and no response has been received, the client resends the request to the same server up to a maximum number of attempts. When the response of a server is received (TCPIP event) the temperature is read and if it differs more than two degrees from the desired level, the thermostat is turned on. The client starts a new cycle, when all servers have sent measurements or the maximum number of attempts has been reached.

5.4.2 Application of the BIP design flow

We focus now on the individual steps of the design flow in Figure 5.2, for the outlined SCC application.

Step 2: synthesis of the OS/kernel model

The case study was based on the detailed modeling of the Contiki OS by taking into account all kernel interactions with the application layer and the network. This allowed us to deliver the first OS kernel library [130].

Step 3: construction of the System Model

The generated BIP system model for the case study (4850 lines of code) consists of 30 atomic components for the AppModel and 26 components for the ConKernel; it includes 430 connectors and 805 transitions. The time step of the model was based on the transmission time per bit through the Contiki network stack. This is given as the inverse of the data rate of a Contiki network access point. The smallest transmitted data unit is one symbol (4 bits) and its transmission

[3] To ease readability, a simplified view of the behaviour is shown from that derived as in Figure 5.7. Guards are shortened (e.g. *POLL* instead of *ev==POLL*) and transitions not relevant to the application logic are omitted (e.g. the *EXIT* event handling, the *msgSnt* and *timerSet* transitions of the templates of Figure C.2 that respectively follow *sndMsg* and *setTimer*.

time is:

$$symbolPeriod = \frac{4}{dataRate} \tag{5.1}$$

For an access point to a wireless medium operating at the 2.4 GHz band, the data rate is 250 kbps and from (5.1) the symbol period is $16\mu s$. Thus, our timing abstraction ignores delays smaller than the inverse of this data rate, which is $4\mu s$. This adjustment allows for a much more fine-grained timing model compared to the one of the Cooja simulator, which is in the *ms* scale. Every parameter of the model that is associated with a time delay can be quantified with an adequate degree of fidelity.

Step 5: state-space exploration

IoT applications are prone to event scheduling delays. Thus, clients set deadlines for expected responses before re-sending the requests. Each deadline is tuned, such that it is feasible to receive a response and avoid sending redundant requests. For the case study, the following functional requirements were formalised as qualitative properties that were checked by state-space exploration:

Functional Requirement FR1 The client reaches `PROCESS_END`.

FR2 The client never sends redundant messages to a server.

FR3 The client collects measurements from each sensor at least once in a specified period.

FR1 specifies a liveness property stating that the client shall terminate. The property is violated if the process does not exit by itself and neither receives an EXIT message from another process, i.e. the corresponding actions are omitted in the code or they are not reachable at runtime. For FR2, a deadline must be set, in which the client obtains a response and avoids sending redundant requests. Every such request is monitored by the observer automaton in Figure 5.10; the property may not be satisfied due to node communication and event scheduling delays that invalidate the set deadline. FR3 implies finding a period for sensing the environment, in which the client will have collected all necessary measurements.

The scenario considered for the state-space exploration involved a limited number of messages to be sent by the two client processes. Specifically, two GET requests, a registration and a de-registration request are sent to each server. Moreover, the GET requests could be attempted twice. The above scenario is sufficient for the generation of all the necessary interaction interleavings that are relevant to the verification of requirements FR1 to FR3.

Step 6: calibration

Calibration was performed based on the execution of the generated code. The temperature distribution was fitted using several measurements taken at random instants during the course of a day. For the motion detection, we fitted a normal distribution with mean $\mu = 1.5$ Volts[4] and standard deviation $\sigma = 1.5$ Volts (we used the distribution fitting method in [133]).

The System Model was augmented with timing information for computations and communications measured in executions of the system. Specifically, (i) the time for the packets' IP header compression/decompression was measured by linking the Contiki OS with the 6LoWPAN protocol implementation of the HC1/HC2 encoding mechanisms [134], whereas (ii) the time for pre- and post-buffering of each message transmission was measured by locating a message and copying its fields from/to the buffers[5]. These time costs are fixed, because the computations are applied to a fixed size input and they do not interfere with other computations until they finish.

Steps 7 and 8: statistical model checking and fault injection

A key functional requirement for an SCC application is to achieve its ultimate control objective:

[4] Motion detection is sensitive to the distance, which impacts the sensor's voltage level. Here we observe the current voltage level of the sensor and not its binary output. A threshold 0.5 V was used with the sensor generating a motion event when the current voltage level is higher than the mean value augmented by the threshold (1.5 + 0.5 = 2V).

[5] The model's time step was much larger than the time taken to store the messages in the transmission/reception buffers.

FR4 Room temperature is maintained within [-2,2] C degrees difference from a user-defined level.

FR4 depends on the temperature variability of the system environment in combination with the client's ability to collect data sufficiently often to act on time (data collection is prone to delays). This requirement was formalised as a stochastic temporal property in PBLTL [135] and the property was validated with the SMC-BIP tool. Moreover, the following key non-functional requirements refer to the performance of the building automation system:

Non Functional Requirement NFR1 Rapid detection of movement during non-working hours, based on the PIR's voltage level.

NFR2 Memory saving by properly sizing the message buffers used by the Contiki network stack in each node.

NFR3 Avoidance of overflows in the asynchronous event queue of each node.

NFR4 Relatively low collision rate in the channel, in order to avoid extensive latencies, which deteriorate the network performance and increase the probability of packet losses.

The mentioned non-functional requirements were formalised as stochastic temporal properties in PBLTL, which were accordingly validated on the calibrated BIP system model.

Regarding the fault injection (step 8 of Figure 5.2), we used the FaultHandler component in Figure 5.11 for studying the system tolerance to extensive bandwidth loss. The FaultHandler was parameterized using probabilistic distributions derived from the analysis of debugging traces [130] obtained from executing the code on the node topology of Figure 5.5.

5.4.3 Domain Specific Language for Contiki REST applications

The DSL for our model-based flow allows specifying a REST Contiki application in a single design definition file. This file is used as input to auto-generate both the BIP model and the C code to be deployed on Contiki nodes; in essence, it ensures the consistency between the BIP model and the application code across the design flow of Figure 5.2.

The language constructs used in REST Contiki applications are encoded into XML elements, and include essential control flow constructs, as well as actions for node communication and event scheduling. All these affect the validity of important functional and non-functional requirements for resource-constrained applications. Each element corresponds to a BIP code template and a template of Contiki C code. The BIP model and the C code are generated through a structure-preserving translation. Thus, every single part of the application model (port, component, data) can be traced to an XML element.

A REST Contiki application is represented by a <RESTapp> element (Listing 5.1) enclosing <module> elements (+ denotes one or more elements). A module contains client and server processes that will be loaded in one Contiki node and is identified by an *id*.

```
<RESTapp>
    <module>+
        (<client>  | <server> )+
    </module>
</RESTapp>
```

Listing 5.1: DSL syntax for a REST Contiki application

A client process is encoded as shown in Listing 5.2 and includes optional *poll* and *exit* handlers (? = 0 or 1 elements), a block of initial actions (*init*) and a repeating *body* of actions. The body includes *wait* actions that enclose event handlers (*onEv* elements) for specific event types. A *wait* action causes the process to block until an event is received. If an incoming event is an EXIT or POLL or if it matches one of the enclosed event types, the corresponding handler is invoked.

Listing 5.3 shows the template that generates the Contiki code for a client process. The template contains placeholders for the code of the process's main elements (e.g. *exit*, *poll* and *body*). The *wait* is represented with a

```
<client>+
    <poll> action+ </poll>?
    <exit> action+ </exit>?
    <init> action+ </init>
    <body>
        ( action*
            <wait>
                <onEv evType="MSG">*
                    action*
                </onEv>
            </wait>
            action* )+
    </body>
</client>
```

Listing 5.2: DSL syntax for client

```
PROCESS_THREAD() {
 PROCESS_EXITHANDLER(/* exit */)
 PROCESS_POLLHANDLER(/* poll */)
 PROCESS_BEGIN();
 . . . . . . /* init */
 while (true) { /* body */
        . . .
        PROCESS_YIELD();
        if (ev == MSG){
                . . .
        }else if(...){ .... }
        . . .
 }
 PROCESS_END(); }
```

Listing 5.3: Contiki code template for client

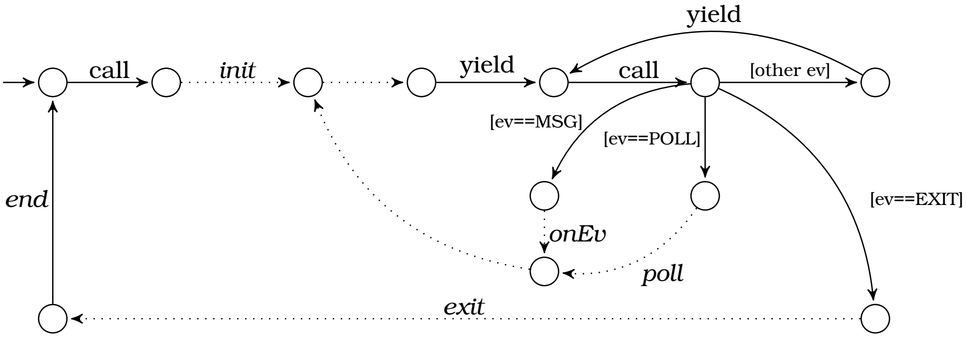


Figure 5.7: BIP template for the client (all event handlers of Listing 5.2 are composed in a single automaton)

PROCESS_YIELD blocking statement and alternative if branches for the enclosed event handlers.

The BIP component for the client is generated using the template in Figure 5.7. After the component starts (*called*), it performs the *init* actions and proceeds with the body loop. The *wait* is represented by: a) a *yield* port leading to a waiting state, b) a *call* port receiving an event, c) internal transitions (executed atomically with their preceding transition) that begin each event handler, d) an internal transition that returns to the waiting state, if the event doesn't match a handler. When the handler is finished, the component performs actions that follow the *wait* and eventually repeats the body. If an exit handler was activated, the component exits the body (*end* port), after the handler has finished.

More details for the DSL syntax are provided in Appendix C.1. Additionally, a complete design of an IoT system includes the mapping of application modules to the system nodes. The DSL syntax for defining this mapping is shown in Listing 5.4; this information is used for generating the BIP system model (process step 3 of Figure 5.2).

Each node's network configuration (*network-config*) determines a set of parameters, which are defined in our XML-based specification [131] (an example file is given in [130]), with default values that may be overwritten in auto-generated Contiki header files (e.g. `uipopt.h`). The same values are also used for the parameterization of the node's network stack model in BIP. For example, one such parameter is the maximum backoff exponent, which influences the network's waiting time, before another attempt to occupy the channel after a collision occurrence. The complete list of configurable parameters is provided in Appendix C.3.

```
<architecture>
    <node ip="string" module="QName">+
        network-config?
    </node>
</architecture>
```

Listing 5.4: DSL syntax for application deployment

5.4.4 BIP models for Contiki WPAN systems

Figure 5.8 outlines the BIP model structure for Contiki WPAN systems. In overall, the System model comprises two layers, the REST Application Model (AppModel) and the Contiki Kernel (ConKernel). The Application Model consists of BIP components for the REST modules that define the client-server interactions and their constraints, as they are derived from step 1 of Figure 5.2. Based on the REST modules mapping to Contiki nodes, the System Model integrates the application model with the BIP components from the OS kernel library and the network stack, for communication through the channel. For the lower level hierarchy we use phrase structure rules, where each rule refers to a BIP compound (shown in "<" and ">") in the left and its constituents (enclosed components) in the right part:

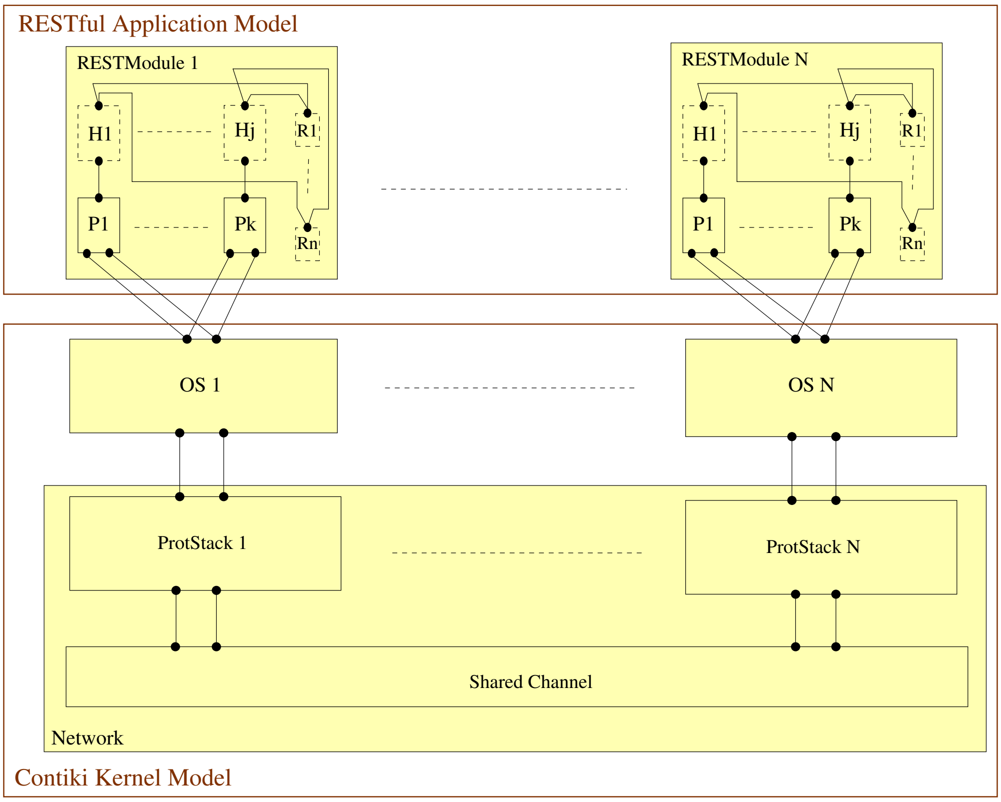


Figure 5.8: BIP model structure for Contiki WPAN systems

⟨*SystemModel*⟩ ::= ⟨*AppModel*⟩ ⟨*ConKernel*⟩
⟨*AppModel*⟩ ::= ⟨*RestModule*⟩+
⟨*RestModule*⟩ ::= Process+ (Resource ResHandler+)*
⟨*ConKernel*⟩ ::= ⟨*OS*⟩+ ⟨*Network*⟩
⟨*OS*⟩ ::= Scheduler Timer CommHandler
⟨*Network*⟩ ::= ⟨*ProtStack*⟩+ Channel
⟨*ProtStack*⟩ ::= MsgSender MsgReceiver

The <AppModel> consists of several <RestModule>, which contain processes, REST resources and their handlers. <ConKernel> includes one <OS> component for each node, and the <Network>, which encompasses components for the network stacks of each node and the communication medium (channel). The interactions between <RestModule> and <OS> are detailed in Appendix C.2. <OS> handles the scheduling of communication, within a node and among nodes. Specifically, it includes a Scheduler, a Timer and a CommHandler [130]). The granularity of behaviour in components ensures that all interleavings of events in a DSL definition are taken into account.

The Scheduler maintains a FIFO queue with the posted asynchronous events, boolean flags with poll requests and a call stack with the active processes, i.e. those that were called and subsequently paused after having posted a synchronous event. When a synchronous event has been handled, the call stack is used to transfer the control to the right active process. A cycle is initiated periodically (period $p_{scheduler}$), in which the Scheduler first sends the requested poll events and then dispatches an asynchronous event from the FIFO. The cycle is completed, when the call stack is emptied. If the queue is full, an error is returned to the process.

The Timer receives timing requests from <RESTModule> processes and stores them in a stack. All requests have a mode, which allows setting, resetting, restarting and stopping a timer. The time advancing is modeled through an interaction that synchronizes all model components. The granularity of a time step affects the simulation efficiency and analysis scalability, and it is therefore determined by a configurable parameter. The remaining time for the next timing interaction is computed separately for each component, and the time advances directly by the minimum number of time steps amongst all components [130].

The CommHandler models the TCP/IP processing and interacts with <Network> (packets are stored in a transmission buffer *TxBuffer* or a reception buffer *RxBuffer*). <ProtStack> includes the *MsgSender* (Figure 5.9) and *MsgReceiver* components, for message transmission (resp. reception) with the CoAP or HTTP protocol. Channel (Figure 5.9) implements behaviour for message transmission and for resolving collisions in simultaneous transmission requests.

The network stack's performance is modelled by proper parameterization of the <Network>. We have the fixed and the modifiable parameters (with default values), shown in Appendix C.3.

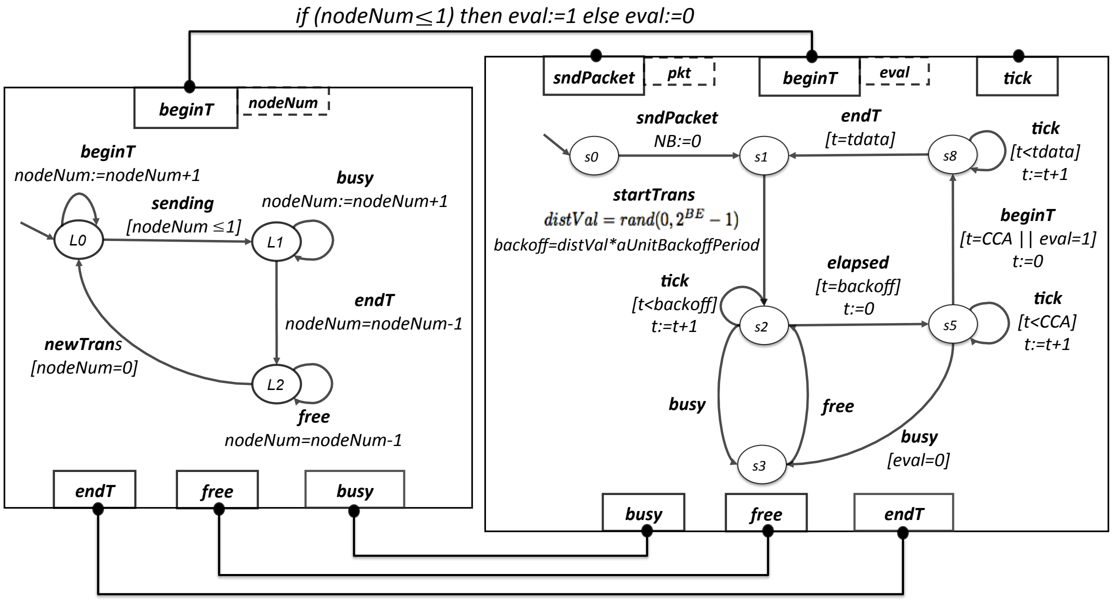


Figure 5.9: Example BIP components and their interactions - Channel (left) and ProtStack.MsgSender (right)

5.4.5 Calibration

The System Model omits characteristics, which can be measured only during the code execution on the nodes and within the system environment. This information includes the characterization of sensor data and HW/SW performance factors, like the time or other resource costs to be quantified at runtime. In the former case, we need to derive probabilistic distributions that fit to measurements of sensor data. In the latter case, it is necessary to employ performance characterization methods, such as the *process profiling* technique [130] that allows measuring isolated blocks of code. The values obtained from this analysis are injected as parameters to the System Model and we eventually obtain the Calibrated System Model (Step 6 of Figure 5.2). We omit the specifics of this step, which is explained in [130], as it falls out of the scope of this thesis.

5.4.6 State-space exploration

For the state-space exploration, each property is formalised as an observer automaton [37] monitoring the events in the BIP model that are relevant to the property, i.e. a set of interactions. Every such event triggers a state transition, which may cause the observer to reach an error control location with no outgoing transitions. If the error location is reached, the property is violated. Observers are attached to the model using broadcast connectors, which allow the model's components to trigger the observers by excluding the other way round. Thus, observers do not interefere with the model and our analysis is sound. To avoid state explosion, it suffices the exploration to be limited to execution scenarios that generate all relevant event interleavings for the examined properties.

As an example, Figure 5.10 shows the observer automaton for the requirement: *The client never sends redundant messages to a server*. The error is reached from *s*2 and *s*3. In *s*2, the client has sent a request (*Cli_sndMsg* port) that has been acknowledged (*NetwCli_recvAck* port), while the server has not yet transmitted (*NetwSrv_transmit* port) or aborted the transmission (*NetwSrv_endSnd* port) of response. In *s*3, the response has been transmitted to the client, but it has not yet been received (*Cli_getMsg* port). Any request sent while in one of these two locations is considered redundant.

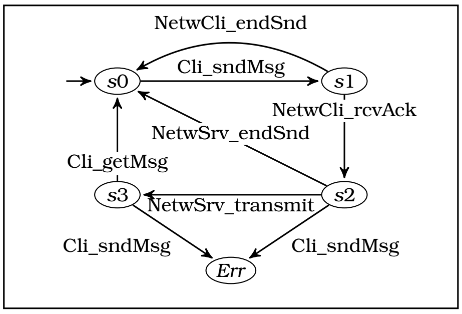


Figure 5.10: Example observer automaton for the formal verification of a qualitative property
(no redundant service requests)

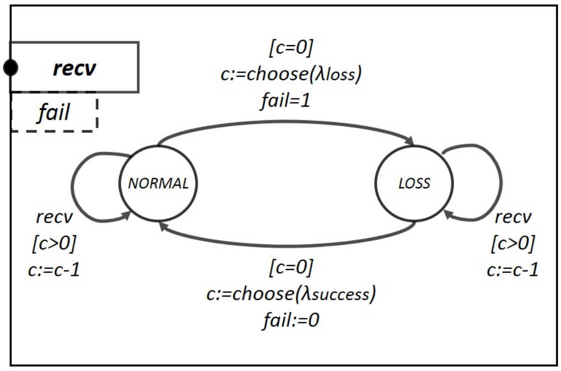


Figure 5.11: Example FaultHandler automaton - packet is ignored if fail = 1

5.4.7 Fault injection

The fault injection (Step 8 of Figure 5.2) is used when there are requirements for the robustness of the IoT system. One such requirement is the tolerance of extensive bandwidth loss, which assumes fault behaviour for studying the impact of consecutive long packet delays or packet losses.

As an example, Figure 5.11 depicts a FaultHandler component that receives the transmitted packets (*recv* port) and decides whether they will be delivered to their destination. This choice is handled through the *NORMAL* and *LOSS* locations, which represent the successful transmission and the case of delayed or lost packets. The FaultHandler remains in each location, for as long as the number of consecutive transmissions is positive. This number is chosen from two probabilistic distributions, $\mathcal{D}_{success}$ and $\mathcal{D}_{loss}$. The FaultHandler is added to the Calibrated System model, which is then validated using SMC.

5.4.8 Experiments and results

The state-space exploration took place within the Contiki 16.04 environment running on a workstation with an Intel i5-3230M CPU@2.60GHz (4 processors), 6GB RAM and 500GB HDD. The SMC experiments were conducted within the Instant Contiki 2.6 environment running in a virtual machine with one CPU core, 1GB RAM and 9GB hard drive.

FR1 was verified as a bounded liveness property using an observer automaton that monitors whether the client has reached `PROCESS_END` before the system model terminates. For FR2, we checked whether the property holds for the temperature monitoring process, using various deadlines (112ms, 160ms and 800ms). The requirement was satisfied for a 160ms deadline, if the motion detection process was not observing simultaneously and it was violated for a 112ms deadline. These results show that the it is less likely for the client to get responses within short deadlines, thus it sends more redundant requests. However, it was possible to get all responses at short deadlines, provided that the motion detection process was not simultaneously consuming network and node processing resources. The FR3 was satisfied for a period of 5.6s, even if the motion detection was operating.

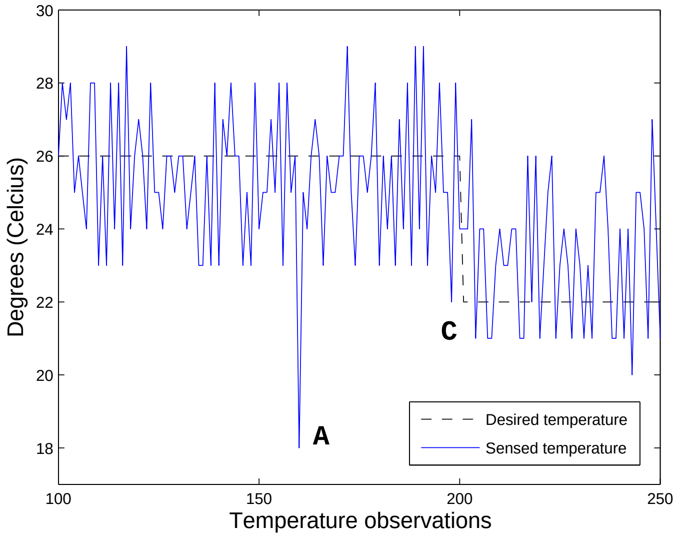


Figure 5.12: Temperature observations (in Celcius) for our building automation application

For FR4, we checked $\phi_1 = |RecvDegree - InputDegree| \leq 2$, where *RecvDegree* is the temperature sensed by the ZIG001 sensors and *InputDegree* is the desired temperature level. We found that $P(\phi_1) = 0.6$, due to the zone-controller responsiveness to temperature changes and fluctuations attributed to external factors. Figure 5.12 shows part of the obtained observations, with the temperature often exceeding the acceptable range, like in point A. Such a rapid change might be attributed to an abrupt change in the environment, as the opening of a window. In point C, the desired temperature has been changed, the zone-controller has perceived the change and the temperature was then reduced by the thermostat.

For the non-functional requirements NFR1 - NFR4, we analysed two sets of execution scenarios: the first set was obtained using the Calibrated System Model (step 7 of Figure 5.2) and the second set by including the FaultHandler component of Figure 5.11 (step 8 of Figure 5.2).

For the second set of execution scenarios, Figure 5.13 shows the transmission time of messages for all servers upon changes in the motion resource. These times are classified in three categories (shown in different colours), namely the minimum observed, the average and the worst-case time. We observe that the

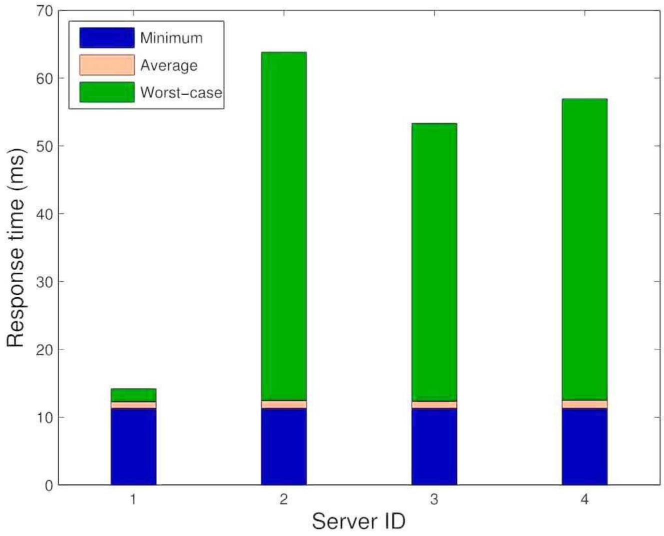


Figure 5.13: Transmission times (ms) for motion detection with faults injected in the Calibrated System Model

worst-case times are significantly different from the two other times, i.e. when there are no collisions or message delays, except from Server 1. This happens because Server 1 does not encounter additional transmission delays, since it is the first to which the zone-controller sends the room's temperature and there are no additional messages to transmit/receive, before the motion resource change messages.

The SMC experiments for NFR1 - NFR4 generated the following results:

NFR1 We checked the property $\varphi_2 = T_{PIR} \leq T_{trans}$, where T_{PIR} is the worst-case message transmission time for the motion resource and T_{trans} the period for a regularly transmitted message, i.e. a client request for the temperature resource (1s). In the first set of experiments, T_{PIR} did not exceed 32 ms, hence $P(\varphi_2) = 1$. In the second set with the presence of the FaultHandler, a higher number of packet collisions occurred with T_{PIR} being approximately 63ms (server ID=2 in Figure 5.13). Nevertheless, the property still holds, since T_{PIR} remains smaller than T_{trans}.

NFR2 We checked the property $\varphi_3 = (size(RxBuffer) < B)$ in the first set of experiments, where B is a bound for the reception buffer size of the protocol stack[1]. B

[1] The bound on the reception buffer size can be adjusted by the parameter *MAX_NUM_QUEUED_PACKETS* of the Contiki kernel; the default value is 2.

depends on $p_{scheduler}$ (cf. Section 5.4.4), which affects the rate in which the MsgReceiver removes messages from the buffer. When $p_{scheduler} = 10\mu s$, $P(\varphi_3) = 1$ if $B = 2$. When $p_{scheduler} = 10ms$, B would have to be 10, so that $P(\varphi_3) = 1$. Thus, $p_{scheduler}$ must be small enough to avoid adjusting the reception buffer size.

NFR3 We have analysed the property: $\varphi_4 = (size(FIFO) < MAX)$, where $size(FIFO)$ represents the size of the Scheduler's FIFO queue and $MAX = 10$. This property holds true ($P(\varphi_4) = 1$) in both sets of experiments.

NFR4 We analysed the property: $\varphi_5 = (NC \leq 1)$, where NC is the number of re-transmissions following a collision. In the first set of experiments we had $P(\varphi_5) = 0.75$, which implies a limited number of collisions. In the second set of experiments, we observed a significant collision rate that resulted in $P(\varphi_5) = 0.5$.

5.5 Discussion

5.5.1 Benefits of the BIP design flow

The design flow of Figure 5.2 supports the progressive development of IoT WPAN systems using BIP models that capture both functional and non-functional system aspects. BIP is component-based, which is essential for enhanced productivity through reuse of model artifacts. The building automation case study in Section 5.4 aimed to test the effectiveness of our approach. Through the invested effort, it was possible to deliver tools (DSL and code generator) and a component library with model fragments for the Contiki OS and the network stack. Table 5.1 reports figures for the required effort and the size of the model and code artifacts for the case study. Each row refers to a particular step of the design flow, but the effort for steps 2 and 4 includes development work, which was done once for the tools and the library of components but can be reused in similar projects.

Table 5.2 reports statistics on the complexity of the generated models. A substantial difference appears between the complexities of the models for the Application and the OS/kernel (*ConKernel*). This is due to the detailed modeling

Step of Figure 5.2	**Effort**	**Scope**	**Product**	**Code (lines)**
1. App design definition	4 days	Application-specific	DSL	120
2. Network configuration	6 hours	Reusable	XML	70
2. IoT comp. lib. & *ConKernel*	7 weeks	Reusable	BIP models	2530
3. App mapping & transform.	4 hours	Application-specific	DSL	40
4. Code generation	8 weeks	Reusable	C	1168
6. Calibration	4 days	Reusable	BIP model	70
8. Fault injection	3 days	Reusable	BIP model	70

Table 5.1: Effort and design artifacts for the building automation case study

Model	**Components**	**Connectors**	**Transitions**	**Code (lines)**
Application model	30	130	223	856
OS/kernel model (*ConKernel*)	26	300	582	3924
System model	56	430	805	4780
Calibrated system model	56	430	820	4850
Fault model	5	13	25	70

Table 5.2: BIP model statistics for the building automation case study

Step (Figure 5.2)	**Verified properties**	**Satisfied properties (without faults)**	**Satisfied properties (with faults)**	**CPU time**	**Memory (MB)**
State-space exploration	Deadlock-freedom	✓	N/A	5h 13 min	4500
	FR1	✓			
	FR2	✓			
	FR3	✓			
Statistical model checking	FR4	60%	55%	2h 10 min	956
	NFR1	100%	100%	1h 35 min	720
	NFR2	100%	100%	1h 21 min	630
	NFR3	100%	100%	0h 24 min	780
	NFR4	75%	50%	5h 4 min	918

Table 5.3: State-space exploration and statistical model checking statistics for the building automation model

of the Contiki OS and network stack functionalities in the library of model components. We thus ensure that all relevant events for the analyzed properties are taken into account, whereas the models should be also valid for additional properties that may be of interest in other Contiki WPAN systems.

Table 5.3 presents statistics for the used resources in steps 5 and 7 of Figure 5.2, i.e. the state-space exploration and the SMC of the building automation model. The verification of deadlock-freedom and the properties for FR1 - FR3 consumed 4.5GB of RAM and was completed in 5h 13 min. 'Note that the CPU time for the SMC of the properties for FR4 and NFR1 - NFR4 varies depending on the confidence and precision parameters used for statistical model checking.

A comparison of BIP's SMC efficiency with the Cooja simulation is certainly interesting (recall that the granularity of our model's time step is in the order of a few μs, whereas Cooja's time step is in the order of *ms*). For this purpose, we conducted simulations of the building automation system during office hours, as well as when it is operating over the whole day including office and non-office hours. The results are shown in Table 5.4.

Simulated time	**Cooja simulation**	**BIP SMC**
8h	1h 03min	1h 20 min
24h	3h 27min	3h 02min

Table 5.4: Simulation time for the building automation model

Here, it is important to recall that Cooja works with a fixed time step advance of the simulation clock, while the BIP simulation advances the clock directly to the time of next event, for all system nodes (cf. Section 5.4.4). The reported CPU times for BIP SMC include as many experiment replications as necessary for providing an accurate verdict.

From the system design point of view, the benefits of our approach stem from the motivating principles in Section 5.3; an informative comparison with other design methods is exposed below.

5.5.2 Limitations

The design flow limitations in its present form concern with its applicability scope, the expectations for experience of its users on formal methods, the degree of process automation and some scalability issues related to the model complexity, the size of systems under design and the supported analyses.

The current applicability scope is the Contiki-based WPAN systems, due to the available model fragments in our IoT component library. For widening the application to other Contiki system architectures, it is essential to enrich the component library, as well as to customize the specification file for the network configuration and the translation for the synthesis of the OS/kernel model (step 2 of Figure 5.2). If the focus is extended to other types of applications than Contiki REST, or to other IoT operating systems, then there is need to modify or even create a new DSL. This implies adaptations or respectively new implementations for the translator used in step 1, the model transformation in step 3 and the code generator in step 4 of Figure 5.2.

If the analysis is restricted to the requirements of our building automation system, there is no need for formal method experience by the designers. For additional requirements concerning timing, memory or thermal aspects, there is still need for some basic knowledge on automata, in order to formulate the new requirements in terms of the BIP system model. For requirements focusing on other aspects, say for example the energy consumption, we need to add appropriate power models for quantifying the energy resources used in computations and communications during the execution of system model. Related work in this direction has been presented in [136].

The degree of process automation is, as always, a matter of balancing usability and flexibility concerns in the provided tool support. We opted to limit the automation to steps 1, 2, 3 and 4 of Figure 5.2, while retaining the ability to manually edit the BIP model during steps 5, 6, 7 and 8, in order to have more space for experimentation during our analyses.

Finally, for the scalability to larger IoT systems, it is important to note that our analyses took place for a limited number of nodes. This is due to the detailed representation of interactions within the *ConKernel* model, since it was generated from our component library in step 2 of Figure 5.2. Such detailed modeling is essential for the state-space exploration and for calibrating the model with sufficient accuracy; it allows validating very diverse requirements and opens prospects for additional requirements in future works. Moreover, for IoT systems of a larger size there is no need for a BIP model that will include all of the system's nodes; for the state-space exploration we only need a model with representative instances of those types of nodes, which suffice to generate all relevant interleavings of events for the properties. For far more complex IoT applications and possible SMC scalability issues, the BIP model can be abstracted using automated stochastic abstraction techniques [137] for identifying those events that are significant for the properties of interest.

5.5.3 Comparison with competitive design methods

The design of IoT systems involves rapidly evolving technologies, diverse applications and a fragmented landscape of operating systems/middleware with various development tools. The major incentives for a model-based design process are: (i) the overall design complexity, due to the high heterogeneity of interconnected things and their interaction modes, and (ii) especially for IoT WPAN systems, the limited computational and energy resources. System models allow reasoning about certain properties of the system's behaviour, and serve as a specification that will lead to a physical implementation of the system, which is compliant with the model. They also provide a means for exploration of different design alternatives. The Flex-eWare model [138] is a general purpose component model for distributed embedded systems that aims to unify model driven and component-based software engineering across many different application domains. This is achieved by integrating generic elements in its metamodel that are instantiated by model libraries. All other design methods in this comparison focus on specific IoT system architectures or application domains.

Table 5.5 summarizes the main characteristics of the design/analysis methods found in related bibliography and compares them with the BIP design flow of Figure 5.2. Most approaches are model-based, but only the BIP flow and [139]–[144] are grounded on languages with formal semantics.

The validation of quantitative specifications or the robustness analysis of systems is supported by some methods, but only [143] and the BIP flow support the verification of qualitative properties, with the latter covering all categories. Moreover, most methods do not cover the entire design cycle (including code

Table 5.5: Logical comparison of design-time analyses and methods for IoT

	Focus	**Scope**	**Qualit. propert.**	**Quantit. propert.**	**Robust. analys.**	**DSL & tool**
[127]	cloud-centric IoT	app design	✗	✗	✗	Aneka Platform-as-a-Service
[139]	REST \Contiki or TinyOS systems	model-based, app design, sys. design, code gener., deployment	✗	✗	✗	graphical Matlab tools, network/hw-in-the-loop simulation
[145]	Sense\Comp. \Control apps	model-based, app design, code gener., deployment	✗	✗	✗	DSL, progr. framework generator, runtime, scenario simulator
[146]	Android & JavaSE devices with MQTT	model-based, app design, sys. design, code gener., deployment	✗	✗	✗	DSL, compiler, linker, runtime system
[147]	iSense OS for WSNs	code gener.	✗	✗	✗	DSL
[148]	services for WSNs (inc. REST)	app design	✗	✗	✗	programming language
[149]	services for WSNs (inc. REST)	model-based, app design, sys. design, code gener., deployment	✗	✗	✗	middleware, network simulation
[150]	services for WSNs (inc. REST)	model-based, app design	✗	✗	✗	DSL
[140]	WSN analysis	model-based, performance \dependability	✗	✓	✓	WSN topolo- gy (GUI), tra- ce analysis
[141]	Arduino,Raspberry Pi (POSIX), Robot OS	model-based, app design, sys. design, code gener.	✗	✓	✗	ThingML (DSL), stat. model checking
[142]	WSN analysis	model-based, performance & dependability	✗	✗	✓	DSL, analytical & behavioural simulation
[143]	REST Contiki Android WSAN systems	model-based, service choreography, deployment	✓	✗	✗	D-LITe middleware, DSL, BeC^3 tool (consiste- ncy check)
[144]	REST \Contiki WSAN systems	model-based, functional design, service choreography, deployment	✗	✓	✗	EMMA middleware, mapper (GUI), satisfiability solver
BIP design flow	REST \Contiki WPAN systems	model-based, app design, sys. design, code gener., deployment	✓	✓	✓	DSL, simul., state explor., stat. model checking

generation and deployment) with the notable exceptions of [139], [141], [143], [144] and the BIP design flow. Some methods are limited to the analysis of WSN systems.

In terms of the tools that support the various forms of analysis, most of the approaches in Table 5.5 provide or utilize various simulation frameworks with the notable exception of the BIP flow along with [140], [141], [143], [144], which support statistical model checking and/or other formal analyses. Finally, a noteworthy number of the shown methods aim to the generation of code for REST services running on the Contiki OS or other operating system.

Chapter 6

Conclusions and future work

6.1 Advancements with respect to state of the art

This thesis presented methodologies and tools for rigorous design of systems. Such an approach aims to derive a system implementation from a set of high-level models by applying a sequence of semantics-preserving transformations. The goal is to tackle the complexity of design by ensuring system requirements, while eliminating the need for a-posteriori verification through correctness-by-construction techniques (property enforcement by design, model-based code generation). The three main contributions of this thesis provide new techniques for how to derive and validate a functional application model, from a set of system requirements or from programs written in an application programming language.

In Chapter 3, we proposed a model-based design flow and tool for the formalization and the early validation of system requirements with respect to system design. We have introduced requirement boilerplates that are associated with property patterns, which capture them in formal terms. The overall process is built on top of correctness-by-construction techniques that open a new perspective in the field. The incrementally built design model in BIP provides evidence of design correctness and consistency among requirements, or else, it guides the revision of requirements. Through the analyzed case study, we managed to enforce by design almost all functional system requirements. The key advantage of our architecture-based approach is that verification at system level was avoided. Instead, we needed to verify the safety properties of a small number of architecture styles with limited state space.

Our incremental approach differs from other approaches [47]–[50] in that it shifts the focus from requirements' allocation and system decomposition based on ad-hoc decisions, to the requirements' formalization and their enforcement through formal BIP architectures. These models are directly applicable to the system design model, thus ensuring correctness-by-construction. Also, as opposed to other approaches [26], [27], [60], Requirement Engineer is guided based on the boilerplate to property pattern associations throughout the formalization of properties.

In Chapter 4, we showed an approach for deriving BIP models based on a compositional execution semantics for programs written in workflow languages . Our approach was instantiated for the BPEL web service composition language. Compositionality here tackles the combinatorial problem of defining semantics for each possible combination of nested statements. The execution semantics of each nesting language construct is specified using safety properties for the BIP component that represents it. Properties are enforced using BIP architecture styles. Through a code generation tool that translates BPEL programs into BIP models we applied our semantics definition to arbitrary-sized programs. The is also used for verifying BPEL programs. We verify essential correctness and other functional properties of the application. Verification is only one of the possible uses of our semantics definition. In [87], our model was extended towards enabling the configuration of information flow policies for BPEL processes.

Multiple works reviewed in [82] define formal execution semantics in various formalisms. However, only a few of them feature a tool [94], [100], [102]. The key advantage of our approach is that we focus on the compositional definition, such that it is possible to define each construct's execution semantics independently of its enclosing or enclosed constructs. Moreover, our compositionality is founded on a correct-by-construction methodology, that guarantees translation correctness, with respect to concretely defined requirement specifications.

Chapter 5 introduces an approach for maintaining the consistency between the BIP system model and the application code across the rigorous design steps. To this end, we developed a domain-specific language (DSL), which was used in the rigorous design of resource-constrained RESTful IoT applications for WPAN systems with nodes running the Contiki OS. Our DSL is a means for a design definition of the IoT application and its deployment mapping. This design definition is also used for auto-generating the code to be deployed to the system nodes, thus preserving the properties of the validated BIP models. Our rigorous design approach was applied to a building automation application. We verified functional requirements related to service responsiveness, whereas our analysis also included important extra-functional requirements.

The BIP design flow, compared to other formal model-based approaches [139]–[142], supports the validation of qualitative properties, which is a very important specification category for IoT and distributed systems in general. The proposed DSL language is a novel specification language that allows taking into account the execution semantics of IoT applications that largely depends on components of the running OS.

In overall, the discussed contributions to the rigorous system design approach address the following issues:

- early validation of system requirements and design towards reducing the validation testing during the late stages of development;
- automated generation of functional application models that preserve the semantics of programs written in programming languages with nesting syntax (we focused on BPEL);
- how to maintain the consistency between the system model and the application code across the rigorous design steps using a domain-specific language (we focused on resource-constrained IoT system design).

6.2 Future research prospects

Although we provided evidence that our approaches can drastically reduce the complexity of modern system design, significant challenges remain to be addressed. In future work, we plan to deal with such difficulties that arise from the adoption of rigorous design in a realistic industrial context.

In the context of the architecture-based design from system requirements, we will work on the automatic construction of atomic components for the initial design model. Moreover, we will extend the framework with more architecture styles and a larger set of classes in the conceptual model. Finally, for the representation of domain knowledge, we need to develop a domain ontology, like the one in [151], to allow for a more fine-grained interpretation of requirements. The incorporation of an ontology was left as future work, since building a rich domain ontology requires dedicated work by domain experts.

For the design of workflow programs, which are data-intensive, it is worth to consider using the secureBIP extension [152], as a means for the analysis and synthesis of security configurations that can ensure data integrity and event non-interference. Moreover, it is also possible to extend our translator to handle other composition languages. For instance, by using BIP as a multilanguage host framework we can analyze the behavioral correctness of service choreographies.

Considering the proposed flow for IoT systems, we intend to provide support for additional extra-functional requirements, related to energy consumption [136], [153] and security aspects. For the latter, it has been planned to extend the design flow with BIP components that model security mechanisms of the Contiki OS [154], for being able to identify vulnerabilities and verify the IoT system's protection against malicious attacks, such as node spoofing [155] and denial of service [156].

Appendix A

A.1 Derived Property Patterns

A.1.1 Prefixes

The prefixes that contain events define necessary and sufficient preconditions that *trigger beg(M)*.

States are used in prefixes as additional necessary preconditions that *enable beg(M)*.

P1: if e1, ... From the P1 template, the properties *P1.1* and *P1.2* are derived, expressing that

- the observation of the event enables *beg(M)*, i.e., "*globally, the event should be observed before an observation of beg(M)*, formulated as:

  P1.1: globally, *occ*(e1) precedes *beg*(M)

- the observation of the event triggers *beg(M)*, i.e., "*globally, beg(M) is observed after the observation of the event*", formulated as:

  P1.2: globally, *beg*(M) responds to *occ*(e1)

P3: while s2, ... From the P3 template, the property *P3.1* is derived, expressing that the state is a necessary precondition, i.e., "*beg(M) is observed only whenever the state is observed*", formulated as:

P3.1: between *beg*(M) and **X** *beg*(M),*obs*(s1) exists

A.1.2 Suffixes

Suffixes impose additional constraints to the occurrence of *beg*(M).

S1: ...before *e2* From the S1 suffix, which should be used always in combination with a prefix, the *S1.1* property is derived, expressing that event e2 is a deadline for the occurrence of *beg*(M), i.e.,"*after an observation of the prefix, the event e2 is not observed before beg(M).*", which is formulated as:

S1.1: between *obs*(P) and *beg*(M), *occ*(e2) is absent

S2: ...sequentially From the S2 suffix, the *S2.1* property is derived, expressing that the main specification is executed in a sequential manner, i.e., "*after the observation of beg(M), end(M) is observed before a consecutive observation of beg(M).*", which is formulated as:

S2.1: between *beg*(M) and *beg*(M), *end(M)* exists

A.2 Case study

A.2.1 Functional architecture

- *CDMS status:* CDMS's status reporting to the EPS subsystem
- *HK PL:* HK data generation for the PL subsystem
- *HK COM:* HK data generation for the COM subsystem
- *HK EPS:* HK data generation for the EPS subsystem
- *HK CDMS:* HK data generation for the CDMS internals
- *Payload:* payload operations' management
- *Error Logging:* hardware errors' logging
- *Flash Memory:* data management in flash memory
- *I2C_sat:* communication through I2C_sat bus

A.2.2 Physical architecture

The physical architecture for the case study is identical to the functional architecture (cf. A.2.1).

A.2.3 Initial design model

Figure A.1 shows a high level view of the initial design model. Such a high level design model depicts the component ports and their in-between connectors.

A.2.4 Requirements and properties of the running example

We present here the requirements and the derived properties of the CubETH running example.

HK-02: ‘ *HK_PL shall handle HK data from the PL subsystem every TBD seconds, as long as the handling of HK data is enabled.* ’

P2: if ⟨e1: [TBD] seconds pass ⟩ and ⟨s1: HK for PL is enabled ⟩

M1: ⟨f1: HK PL ⟩ shall ⟨a1: handle HK data from PL ⟩

Derived Properties:

HK-02-P2.1 globally, *occ*(e1)∧ *obs*(s1) precedes *beg*(a1)

HK-02-P2.2 globally, *beg*(a1) responds to *occ*(e1)∧ *obs*(s1)

Attribute values based on the resulting model:

obs(s1): `HK_PL.enabledHK_PL`,

occ(e1): `Environment.HKPL_TBD_pass` ,

beg(a1): `HK_PL.beginHK`

HK-03: ‘ *While the PS for the PL subsystem is not enabled, HK_PL shall transmit the HK data of the PL subsystem through the TC/TM service.* ’

P3: while ⟨s1: PS for PL is not enabled ⟩

M1: ⟨f1: HK PL ⟩ shall ⟨a1: transmit HK data through the TC/TM service ⟩

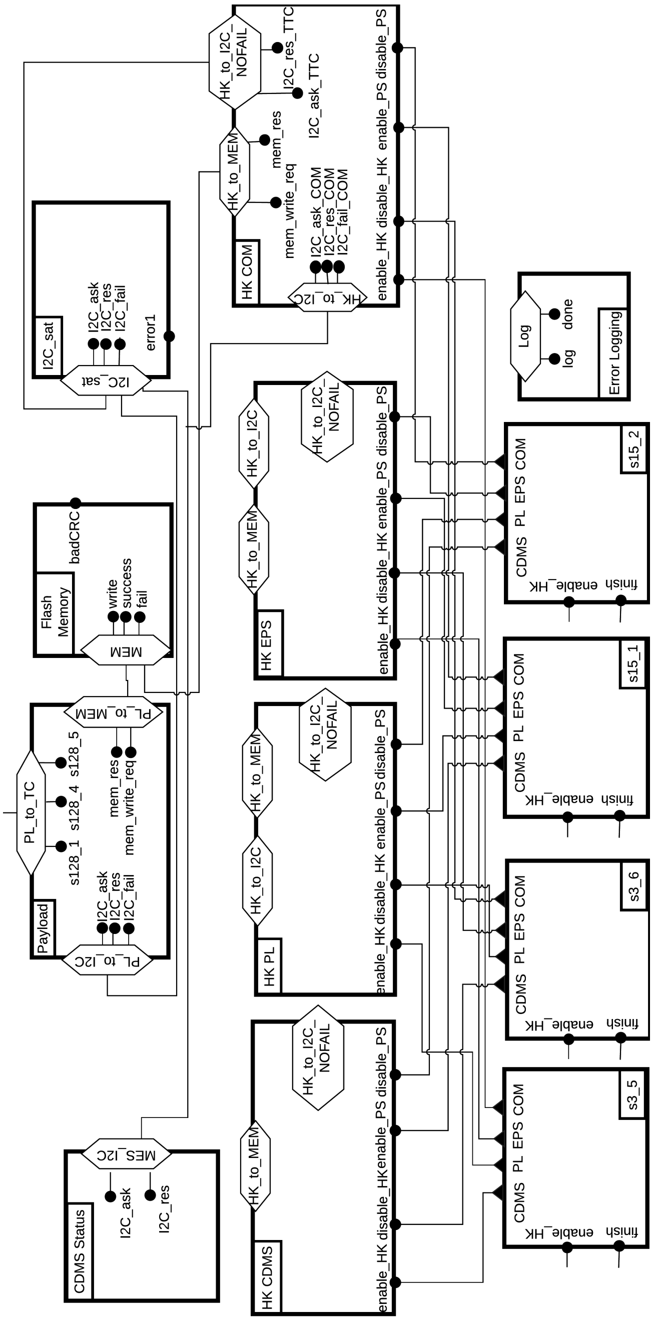


Figure A.1: The high level initial design model for the CubETH case study.

Derived Properties:

HK-03-P3.1 globally, *obs*(s1) precedes *beg*(a1)

HK-03-P3.2 globally, *beg*(a1) responds to *obs*(s1)

Attribute values based on the resulting model:

obs(s1): `HK_PL.disabledPS_PL`,

beg(a1): `HK_PL.ask_I2C_TTC`

HK-04: ‘ *HK_PL shall write HK data to the flash memory, if PS for the PL subsystem is enabled.* ’

P3: while ⟨s1: PS for PL is enabled ⟩

M1: ⟨f1: HK PL ⟩ shall ⟨a1: write HK data to the flash memory ⟩

Derived Properties:

HK-04-P3.1 globally, *occ*(e1) precedes *beg*(a1)

HK-04-P3.2 globally, *beg*(a1) responds to *obs*(s1)

Attribute values based on the resulting model:

obs(s1): `HK_PL.enabledPS_PL`,

beg(a1): `HK_PL.mem_write_req`

HK-05: ‘ *HK_PL shall contact the EPS for a restart of the PL subsystem after a failure persists for [TBD] sec.*’

P1: if ⟨e1: a failure of subsystem * persists for [TBD] sec ⟩,

M1: ⟨f1: HK PL ⟩ shall ⟨a1: contact the EPS for a restart of PL ⟩

Derived Properties:

HK-05-P1.1 globally, *occ*(e1) precedes *beg*(a1)

HK-05-P1.2 globally, *beg*(a1) responds to *obs*(s1)

Attribute values based on the resulting model:

occ(e1): `Environment.HKPL_failurePers`

beg(a1): `HK_PL.I2C_ask_EPS`

and their properties

A.2.5 Final design model

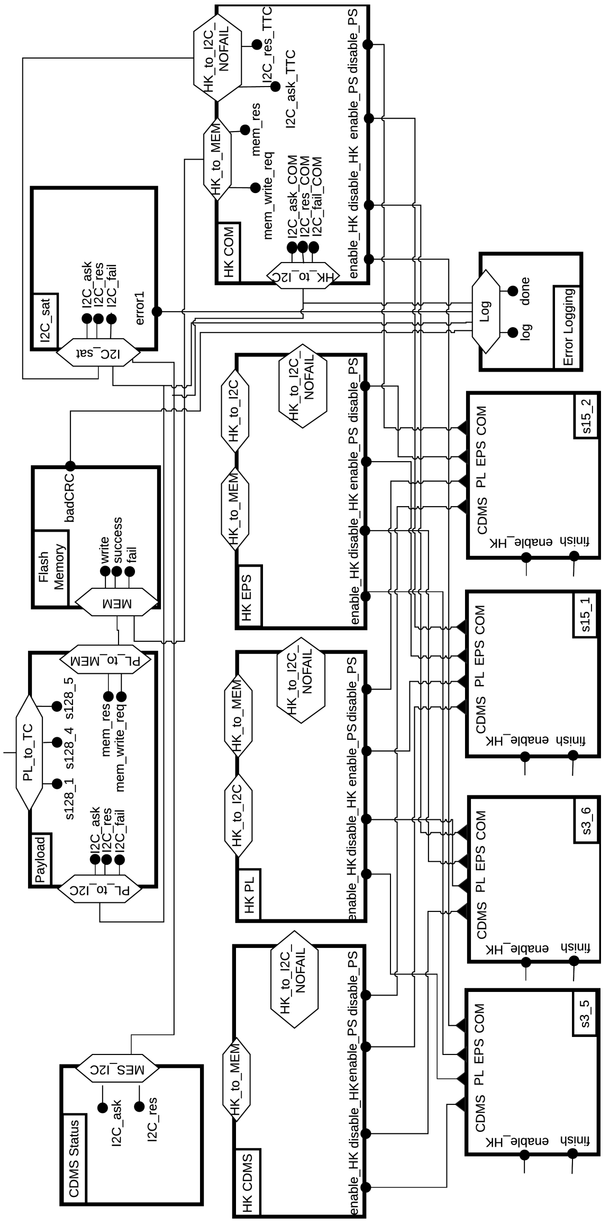


Figure A.2: The high level final design model for the CubETH case study.

In the high level view of the final design model, compared to that of the initial design in Figure A.1, additional connectors have been added for property enforcement. Specifically, these connectors were added between the Error Logging component and other components of the model.

HK PL

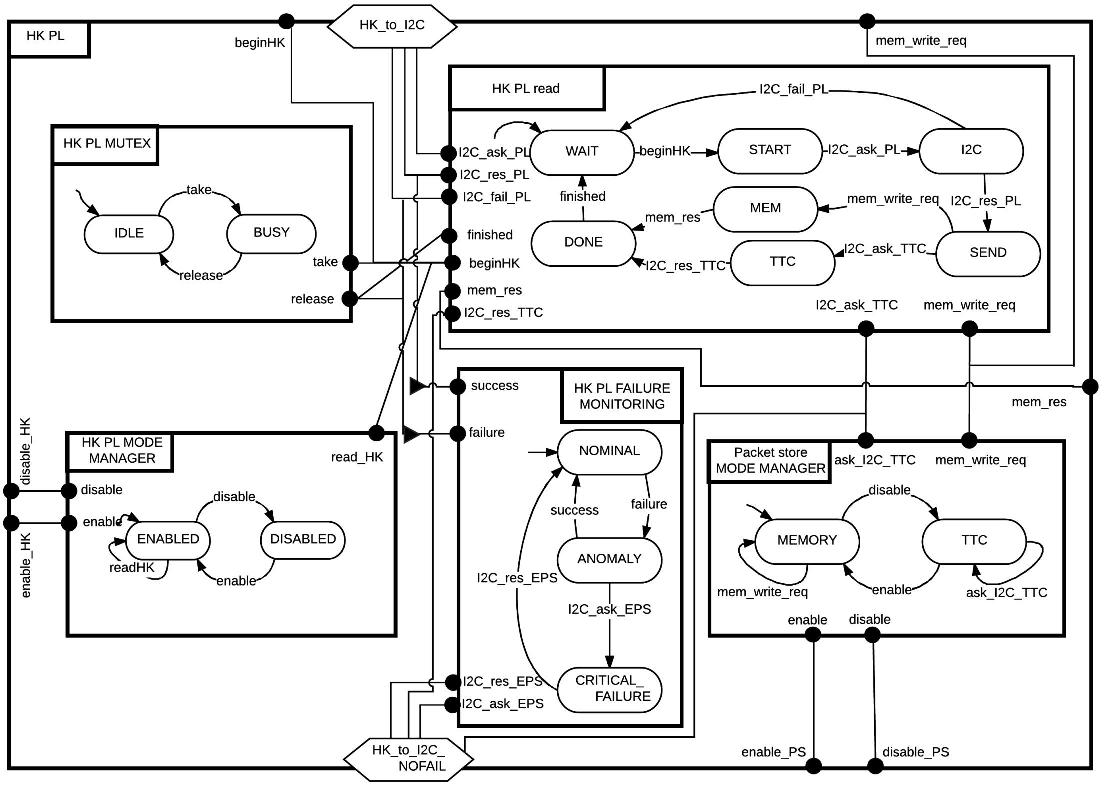


Figure A.3: The HK PL component (The HK COM and HK EPS are also like HK PL)

The requirements for the HK PL component are shown in Section A.2.4.

HK CDMS

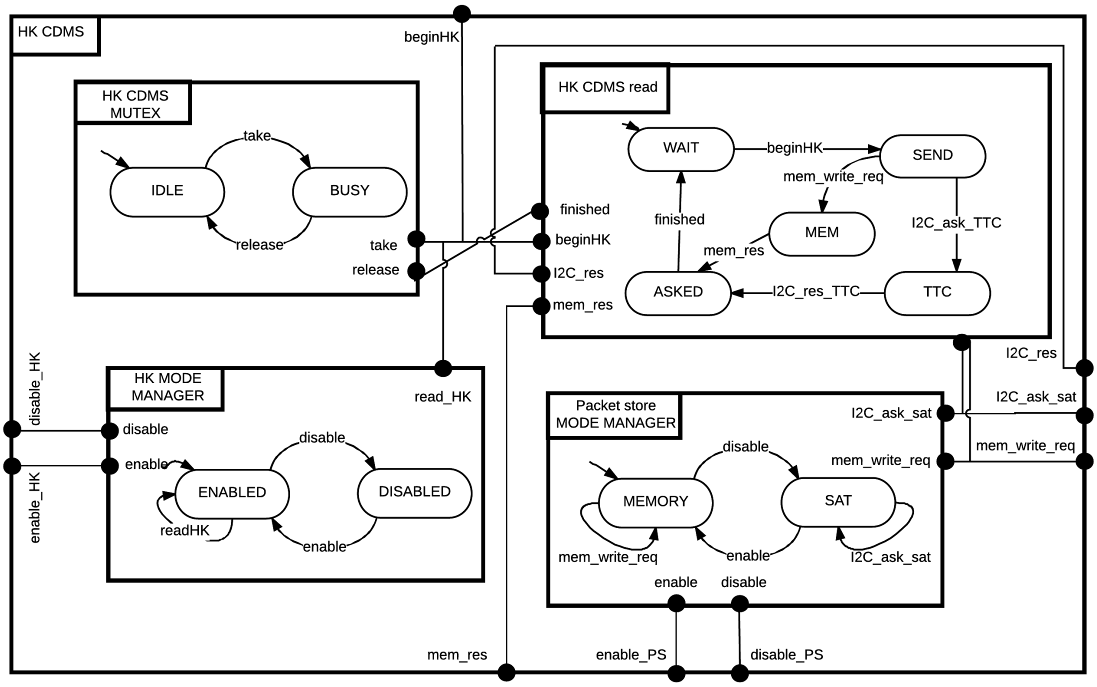


Figure A.4: The HK CDMS component

The requirements for the HK CDMS component are similar to the HK-02, HK-03 and HK-04 requirements (of HK PL component) shown in Section A.2.4.

CDMS status

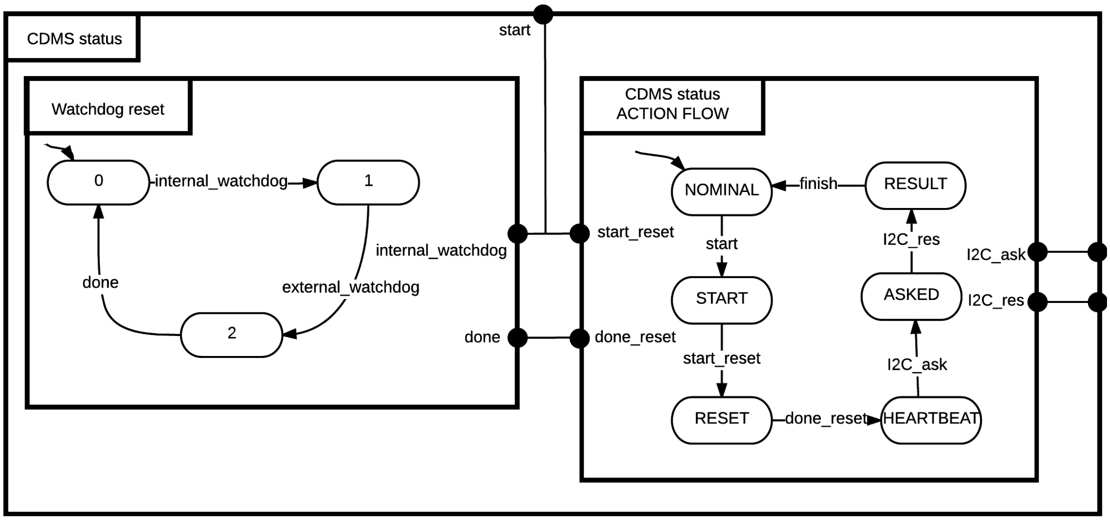


Figure A.5: The `CDMS status` component

CDMS-02: ‘ *The CDMS_status shall periodically reset the internal and external watchdogs and contact the EPS subsystem with a “heartbeat”.* ’

P1: ⟨e1: if [TBD] seconds pass ⟩

M2: ⟨f1: CDMS_status ⟩ shall ⟨a1: reset the internal and external watchdogs ⟩ and ⟨a2: contact the EPS subsystem with a “heartbeat” ⟩

Error Logging

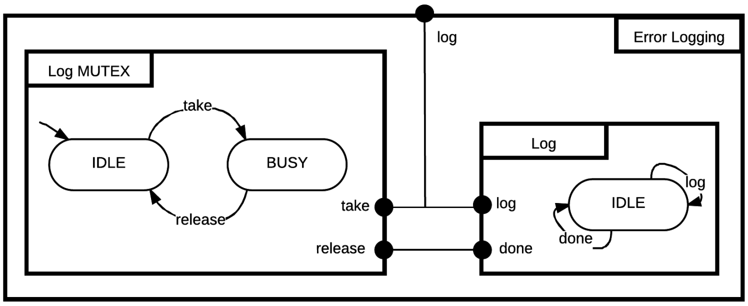


Figure A.6: The `Error Logging` component

Log-02: ‘ *Error_logging shall log each hardware error to the RAM.*’

P1: if ⟨e1: a hardware error is produced ⟩

M1: ⟨f1: Error_logging ⟩ shall ⟨a1: log the error to the RAM ⟩

Log-03: ‘ *Error_logging shall not log two errors simultaneously.* ’

M1: ⟨f1: Error_logging ⟩ shall ⟨a1: log the error to the RAM ⟩

S3: sequentially

Payload

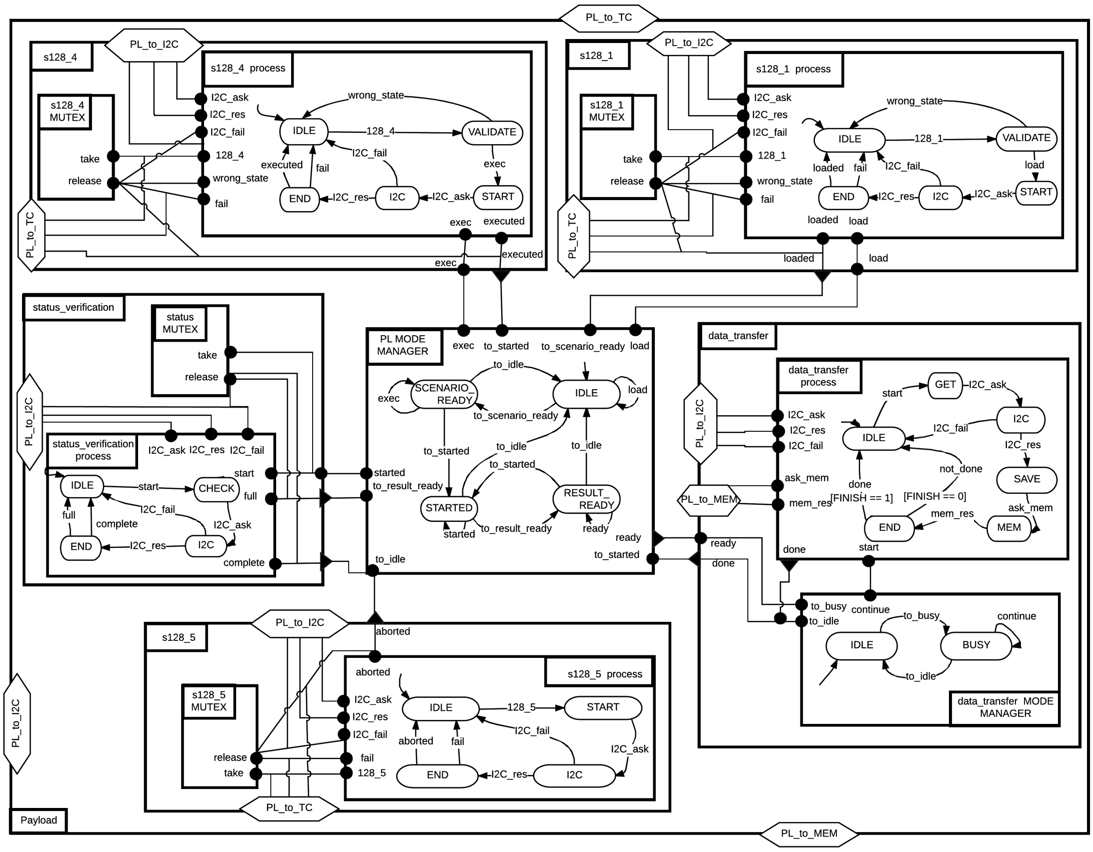


Figure A.7: The `Payload` component

PL-01: ‘ *When in IDLE mode, PL shall load a scenario to the board.* ’

P3: while ⟨s1: in IDLE mode ⟩

M1: ⟨f1: PL ⟩ shall ⟨a1: load a scenario to the board ⟩

PL-02: ‘ *In SCENARIO_READY, PL has loaded a scenario to the board.* ’

P1: if ⟨e1: PL has finished loading a scenario to the board ⟩

M3: ⟨f1: PL ⟩ shall ⟨s2: be in SCENARIO_READY mode ⟩

PL-03: ‘ *In SCENARIO_READY, PL shall execute a scenario to the board.* ’

P3: while ⟨s2: in SCENARIO_READY mode ⟩

M1: ⟨f1: PL ⟩ shall ⟨a12: execute a scenario to the board ⟩

PL-04: ‘ *In STARTED mode, a PL scenario has been executed.* ’

P1: if ⟨e2: PL has finished executing a scenario ⟩

M3: ⟨f1: PL ⟩ shall ⟨s3: be in STARTED mode ⟩

PL-05: ‘ *In STARTED mode, PL shall check the status of the board’s internals.* ’

P3: while ⟨s3: in STARTED mode ⟩

M1: ⟨f1: PL ⟩ shall ⟨a3: check the status of the board’s internals ⟩

PL-06: ‘ *If the board status is full, PL shall be in the RESULT_READY mode.* ’

P2: if ⟨e3: the board status is found full ⟩ and ⟨s5: there is data to be transferred from the board ⟩

M3: ⟨f1: PL ⟩ shall ⟨s4: be in RESULT_READY mode ⟩

PL-07: ‘ *In RESULT_READY, PL shall transfer data from the board to the flash memory.* ’

P3: while ⟨s4: in RESULT_READY ⟩

M1: ⟨f1: PL ⟩ shall ⟨a4: transfer data from the board to the flash memory ⟩

PL-08: ‘ *PL shall be back to IDLE mode, whenever PL aborts a board operation.* ’

P1: if ⟨e4: PL has finished aborting a board operation ⟩

M1: ⟨f1: PL ⟩ shall ⟨s1: be in IDLE mode ⟩

PL-09: ‘ *PL shall not be processing two (128,1) telecommands simultaneously.* ’

M1: ⟨f1: PL ⟩ shall ⟨a6: process (128,1) telecommands ⟩

S2: sequentially

PL-10: ‘ *PL shall not be processing two (128,4) telecommands simultaneously.* ’

M1: ⟨f1: PL ⟩ shall ⟨a7: process (128,4) telecommands ⟩

S2: sequentially

PL-11: ‘ *PL shall not be processing two (128,5) telecommands simultaneously.* ’

M1: ⟨f1: PL ⟩ shall ⟨a8: process (128,5) telecommands ⟩

S2: sequentially

PL-12: ‘ *PL shall not perform two status verification tests simultaneously.* ’

M1: ⟨f1: PL ⟩ shall ⟨a9: perform status verification tests ⟩

S2: sequentially

Flash Memory

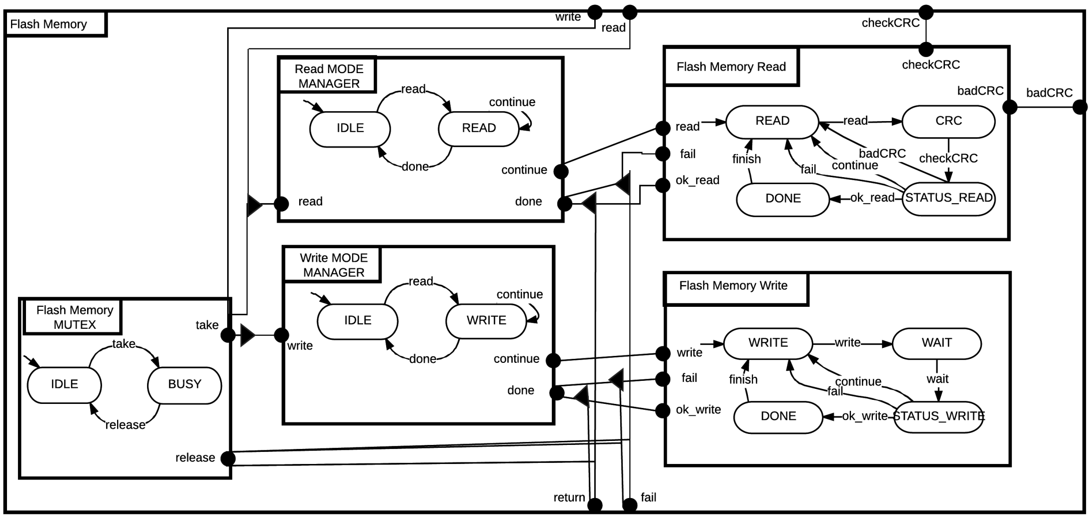


Figure A.8: The `Flash Memory` component

Mem-01: ‘ *Flash memory shall process read and write operations sequentially.* ’

M1: ⟨f1: Flash_memory ⟩ shall ⟨a1: process operations ⟩

S2: sequentially

Mem-02: ‘ *For a write operation, the flash memory writes blocks of data the device, until all data has been written.* ’

P3: while ⟨s1: a write operation is being processed ⟩

M1: ⟨f1: Flash_memory ⟩ shall ⟨a2: write data to the device ⟩

Mem-03: ‘ *For a read operation, the flash memory reads each block of data from the device and performs the Cyclic redundancy check (CRC), until all data has been read.* ’

P3: while ⟨s2: a read operation is being processed ⟩

M2: ⟨f1: Flash_memory ⟩ shall ⟨a3: read data from the device ⟩ and ⟨a6: perform the CRC ⟩

Mem-04: ‘ *Each read operation returns its finishing status.* ’

P1: if ⟨e1: a read operation begins ⟩

M1: ⟨f1: Flash_memory ⟩ shall ⟨a4: return the operation’s finishing status ⟩

S1:before ⟨e4: it has finished ⟩

Mem-05: ‘ *Each write operation returns its finishing status.*’

P1: if ⟨e2: a write operation begins ⟩

M1: ⟨f1: Flash_memory ⟩ shall ⟨a5: return the operation’s finishing status ⟩

S1:before ⟨e5: it has finished ⟩

Mem-07: ‘ *If CRC fails, the Flash memory shall reread the data from the flash memory, as long as the number of read attempts is less or equal to [MAX_FM_-READS].* ’

P2: if ⟨e3: CRC fails ⟩ and ⟨s3: the same data has been read [MAX_FM_-READS] times or less ⟩

M1: ⟨f1: Flash_memory ⟩ shall ⟨a6: continue reading data from the device ⟩

Mem-08: ‘ *If CRC fails, the Flash memory shall abandon the reading operation, as long as the number of read attempts exceeds [MAX_FM_READS].* ’

P2: if ⟨e3: CRC fails ⟩ and ⟨s4: the same data has been read more than [MAX_FM_READS] times ⟩

M1: ⟨f1: Flash_memory_read ⟩ shall ⟨a7: abort the read operation ⟩

I2C_sat

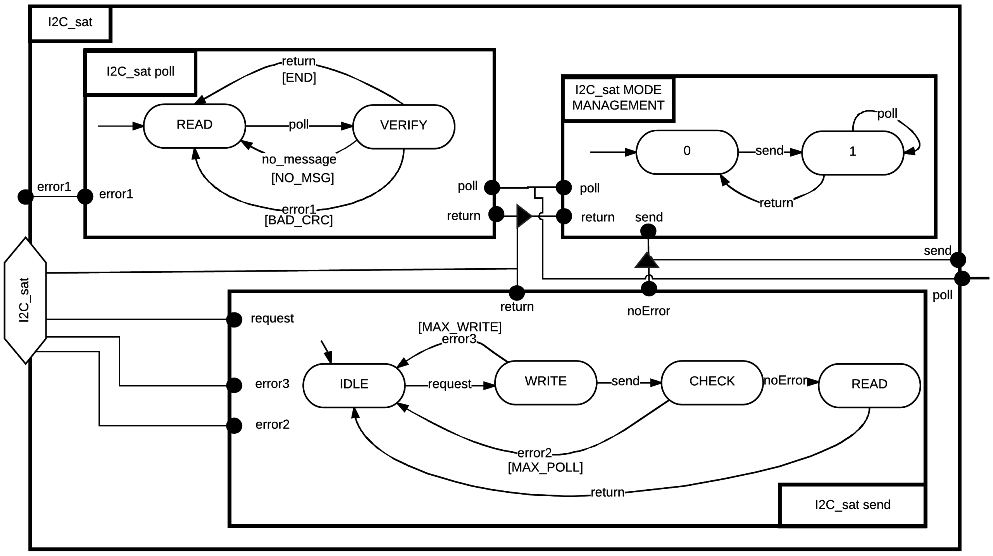


Figure A.9: The `I2C_sat` component

The funcitonality of the `I2C_sat` component is taken into account in the model, though it is not specified in the requirements. The component implements the I2C protocol, which is specified in [41].

Appendix B

B.1 Variables for process state

B.1.1 The state of service interactions

The lifetime of service interactions spans across the execution of multiple activity components. Therefore, we use variables in the `data` components of scopes that allow sharing information about the state of service interactions. These variables store:

- the (url) location of remote services of partner links that influences the routing of incoming messages. Partner links may or may not be initialized (by the `scope` or `copy` components) when they are accessed, in which case a fault is thrown.
- the correlation sets; these are sets of *correlation properties*, i.e. variables that are instantiated and accessed by activity components for service interactions (`receive`, `reply`, `invoke`, `listen`). A fault (correlation violation) is thrown upon attempts to (i) initialize already initialized sets, (ii) access uninitialized sets, (iii) send a message not matching the initialized sets.
- the information for routing messages to each listening `receive` component[1]; this information consists of a partner link, a service operation, as well as a list of correlation set values and their mappings to message parts.

[1] this refers to enabled IMA according to the BPEL terminology

A fault is thrown if the routing information of receive components are conflicting (i.e., they match the same messages) or ambiguous (i.e., they can both match a message).

- a request identifier for each incomplete incoming synchronous request[2]; this identifier is set by the receive component and encodes its associated partner link, service operation and possibly some internal transaction identifier[3]; each reply component attempts to erase the request identifier to which it replies. A fault is thrown if: (i) a receive component attempts to set a duplicate request identifier (conflicting request), (ii) a reply does not find its associated request identifier (missing request), or (iii) a scope about to end detects an incomplete synchronous request (missing reply).

B.1.2 BPEL variables

The variables for a BPEL process may have been defined globally or within scopes. They store the content of messages or any other information that is shared among the activities of a scope, and are often used in conditions that influence the control flow. Their data types are either XML types or WSDL message types with partitions, called *parts*. In the BIP model, these variables are stored within the data components using separate BIP variables for each of their parts; they are read and assigned by activity components such as copy, receive, listn, invoke, reply and loopctrl using their read and write ports.

For the values of BPEL variables, we have adopted a data abstraction approach using symbols. This allows identifying variables that have not been initialized, and assignments with the same expression which is needed to detect every time that the variables change value. The same default symbol is assigned to all variables that have not been initialized. Each activity component with assignment semantics (e.g. copy, receive) evaluates the symbol to be assigned - let us call it right-value - to some BPEL variable, referred to as left-value. First, the symbols used in the right-value are being read:

[2] this refers to open IMA according to the BPEL terminology
[3] this refers to the BPEL messageExchange attribute

- if they correspond to BPEL variables, then they are retrieved from data components;
- if they are BPEL message inputs or external data, then their values are represented by distinct symbols.

To the left-value is then assigned the hash code for the string given by concatenating the retrieved symbols with the static parts of the right-value.

B.2 Compositional semantics for BIP compounds

B.2.1 BIP compound for sequence

Definition B.2.1. A sequence composite encloses n components act$_1$...act$_n$. The following safety properties have to be satisfied:

- if sequence is started, so does act$_1$.
- act$_i$ is started only if act$_{i-1}$ is finished.

The safety properties of Def. B.2.1 and the general properties are fulfilled by the *Sequential* style of Figure B.1. The style has a coordinator P which enforces a sequential order of execution between two parameters A1 and A2. Thus for the coordination of $n > 2$ parameters, *Sequential* style must be applied hierarchically n-1 times. The first and second property of Def. 4.4.1 are implemented by two rendezvous between the P.*start* and A1.*start* ports (resp. of A1.*fin* and A2.*start* ports). Note that *fin* ports are linked by connectors twice in A1: first, in order to synchronize the *fin* and *start* of successive components, and second, to synchronize the *fin* of both components at the end of their processing. The rendezvous between the A1.*rvs* and A2.*rvsd* ports enforce the order of compensation, which is the reverse of the order of execution. In case A1 *abort* ports, it can cause A2 to be disabled by triggering one of the *abort* ports. This coordination, though it is not applied to sequence, it is needed for using the style to other components with similar semantics (e.g. the act composite).

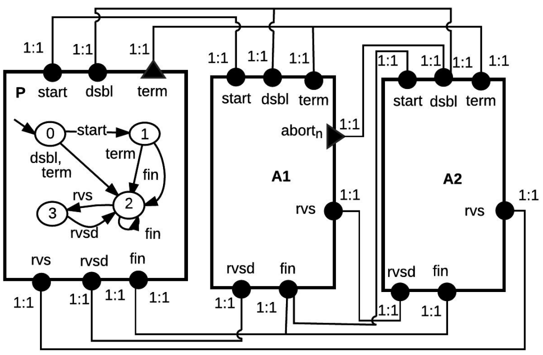


Figure B.1: *Sequential* style

The connectors in Figure B.1 that connect the *dsbl*, *term* and *fin* ports are used to enforce the general safety properties in Section 4.4. These connectors exist also in the Prioritized Alternative, Non-Prioritized Alternative, Repetitive and Parallel Repetitive styles, though they are omitted from the diagrams. The connectors of *rvs* and *rvsd* ports for these styles are placed as shown in Figure 4.6. In all these style there is one coordinating component which is branching bisimilar to the P component shown in Figure 4.6. This ensures branching bisimilarity between the observable behavior of the styles and the empty component.

B.2.2 BIP compound for if

Definition B.2.2. An if composite encloses one condctrl component and n components $\mathtt{act}_1 \ldots \mathtt{act}_n$. Let us consider the condctrl ports $acc_1 \ldots acc_n$ and $rej_1 \ldots rej_n$, for accepting an activity component (resp. rejecting it). The following safety properties have to be satisfied:

- $\mathtt{act}_i$ is started only if it is accepted by the condctrl.
- if condctrl rejects an $\mathtt{act}_i$, then $\mathtt{act}_i$ is disabled.

The *Conditional alternative* style of Figure B.2 is used to enforce the safety properties of Def. B.2.2. The style has one CN and n A parameters. The coordinator P plays the same role as in the previously described styles. The connectors that enforce the general safety properties are omitted from Figure B.2. Parameters A are assumed to be branching bisimilar with empty, whereas CN is assumed to be branching bisimilar to the behavior with which it is depicted in Figure B.2. The first property of Def. B.2.2 is fulfilled by n broadcast connectors between each $\mathtt{CN}.selct_i$ and $\mathtt{A}_i.start$ for $i = 1 \ldots n$. The second property is satisfied by a single broadcast connector between CN.*rej* and all $\mathtt{A}_i$.*disable*. Broadcast connectors were chosen, so that CN is not blocked after the termination of the if composite, upon which all $\mathtt{A}_i$ are terminated while CN doesn't enable termination. As a result, CN must be able to perform, for example, $selct_i$ even if $\mathtt{A}_i$ cannot start. Finally, it is worth to mention that the effects of disabling some $\mathtt{A}_i$ do not depend on race conditions, thus the same results are produced whether $\mathtt{A}_i$ is disabled before, during, or after the execution of some $\mathtt{A}_j$.

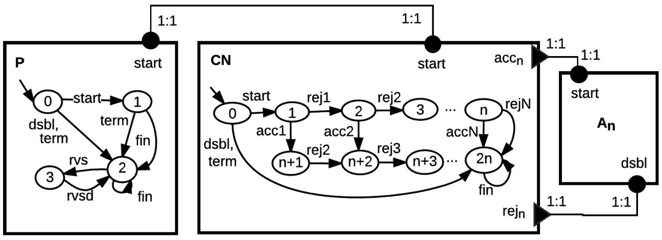


Figure B.2: Conditional alternative style

B.2.3 BIP compound for `pick`

Definition B.2.3. A `pick` composite encloses n components $\mathtt{act}_1 \ldots \mathtt{act}_n$ and n components $\mathcal{L}_1 \ldots \mathcal{L}_n$ (each being either `listn` or `timer`). $\mathcal{L}$ components export the ports *ev* and *off*, for receiving an event (i.e. message or timing event) and, respectively, stop waiting. The following safety properties need to be satisfied:

- if `pick` is started, so do all $\mathtt{L}_i$.
- $\mathtt{act}_i$ is started only if the event of $\mathtt{L}_i$ arrives.
- At most one $\mathtt{act}_i$ is started.

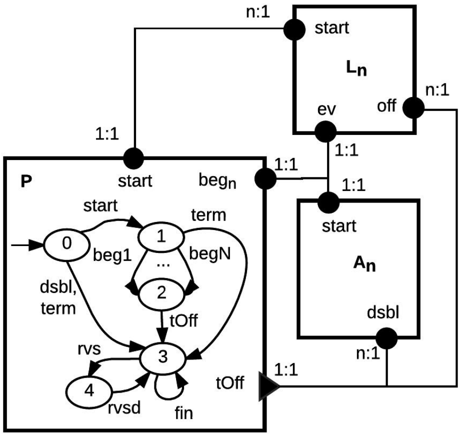


Figure B.3: *Alternative* style

The *Alternative* style (Figure B.3) is used to enforce the safety properties of Def. B.2.3. The style has n `M` parameters, n `A` parameters and a coordinator `P`. The behavior of `L` and `A` is assumed to be branching bisimilar to `empty`. The first property of B.2.3 is fulfilled by a single rendezvous that connects the `P`.*start* and the $\mathtt{L}_i$.*start* ports, while the second is realized by the rendezvous between the $\mathtt{L}_i$.*ev* and $\mathtt{A}_i$ ports.*start*. The third property is satisfied by the coordinator `P` and its rendezvous connection with the $\mathtt{L}_i$.*ev* port, which allows only one such port to be executed. Subsequently, `P` disables the `A` components and closes the `L` components through a broadcast connector triggered by the *tOff* port. Note that the `A` that started will not be disabled, since it is not in the initial state. The broadcast connector allows the interaction to be executed, even if this `A` is not able to participate.

B.2.4 BIP compound for `loop`

Definition B.2.4. A `loop` composite encloses one `loopctrl` component and n components, $\mathtt{act}_1 \ldots \mathtt{act}_n$, standing for n repetitions of the loop body activity. Let us consider the `loopctrl`'s ports $beg_1 \ldots beg_n$, *end* and *break*, for beginning a loop's execution, receiving its end and breaking from loop, respectively. The following safety properties have to be satisfied:

- $\mathtt{act}_i$ is started only if `loopctrl` begins the i–th loop execution.
- if *break* occurs, the $\mathtt{act}_i$ of all remaining loop executions are disabled.

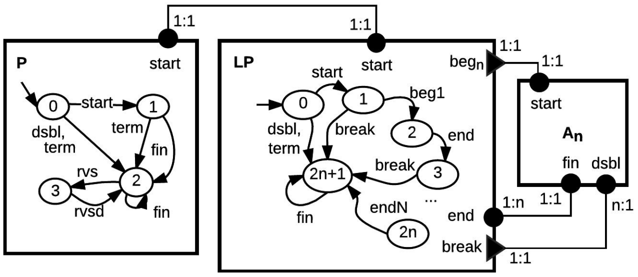


Figure B.4: *Repetitive* style

Note that we use a different activity component to maintain each repetition's state, since the loop's activity compensation will have to run for each repetition separately.

The *Repetitive* style (Figure B.4) is used to enforce the safety properties of Def. B.2.4. The style contains one parameter `LP`, n parameters `A` and a coordinator `P`. The first property is satisfied by the broadcast from `LP`.beg_i to the

$\mathtt{A}_i$.*start* port, whereas the second property is realized by the broadcast from `LP`.*break* to all $\mathtt{A}_i$.*dsbl* ports.

Definition B.2.5. A `parallel loop` composite encloses one `loopctrl` component and n components $\mathtt{scope}_1 \ldots \mathtt{scope}_n$, standing for n parallel repetitions of the loop body `scope`. Let us consider the `loopctrl`'s ports $beg_1 \ldots beg_n$, for beginning i parallel loop executions, and *break* for breaking from loop, respectively. Also, the ports *fail* and *succ* used by `loopctrl` to receive the successful (resp. unsuccesssful) completion of every `scope`. The following safety properties have to be satisfied:

- if `loopctrl` begins i loop executions, $\mathtt{scope}_1 \ldots \mathtt{scope}_i$ are started.
- if `loopctrl` breaks from loop, all terminatable $\mathtt{scope}_i$ are terminated.
- if $\mathtt{scope}_i$ completes, `loopctrl` is notified whether completion was successful or not.

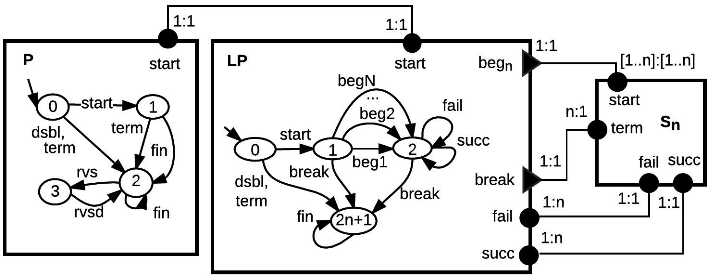


Figure B.5: *Parallel Repetitive* style

The *Parallel repetitive* style (Figure B.5) is used to enforce the safety properties of Def. B.2.5. The style contains one parameter `LP`, n parameters `S` and a coordinator `P`. The first property is satisfied by the broadcast from `LP`.beg_i to the *start* ports of $\mathtt{S}_1 \ldots \mathtt{S}_i$. Number i is evaluated by `loopctrl`. The second property is fulfilled by the broadcast from `LP`.*break*, which is executed by `loopctrl` in 3 cases: (a) if $i = 0$ in state 1 (b) if the number of completed `S` in state 2 is the minimum needed for the loop and (c) if all `S` in state 2 are completed. It depends on the `loopctrl` whether it counts all completed `S` to to the minimum needed, or only those that finished successfully. Finally, the third property is fulfilled by the rendezvous connectors between each $\mathtt{S}_i$.*succ* and `LP`.*succ* (resp. *fail*)

B.2.5 BIP compound for act

Definition B.2.6. An act composite is used to enclose a specific activity component, say acti, one target and/or one source component. Let us consider the target's port *abort*, for preventing the execution of actc due to target links, and the source's ports: $read_1 \ldots read_m$, for reading values used in the evaluation of m source links (also evaluating the links), and $set_1 \ldots set_m$, for setting the evaluation' results to the m links. The following safety properties have to be satisfied:

- actc is started only if (target has finished) and (target has not prevented acti).
- source is started only if acti has finished.
- source evaluates a link only if (acti has not been disabled or terminated) and (neither acti or source has thrown a fault).

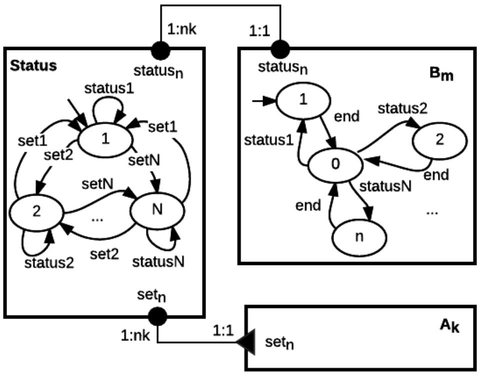
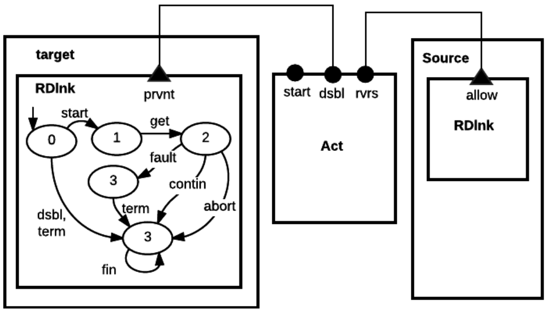


Figure B.6: *Status mngmt* style



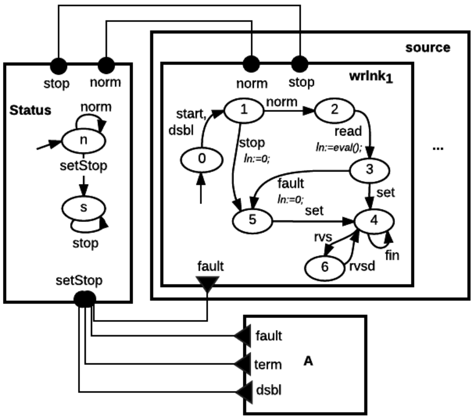


Figure B.7: Reading of links in act component

Figure B.8: Writing of links in act component

For the properties of Def. B.2.6 two architecture styles were combined, namely the *Sequential* and the *Status mngmt* styles. The *Sequential* style is used for the first two properties of Def. B.2.6, that are ensured through the sequential execution of `target`, `acti` and `source` and the ability of `target` to abort the sequence. *Status mngmt* (Figure B.6) is used to enforce the third property of Def. B.2.6. The style has m parameters `B`, and k parameters `A`, which get and set the status, respectively. A coordinator `Status` is used to maintain n status values using a different state for each status value. There are no assumptions for `A`, while `B` are assumed to be branching bisimilar to the behavior shown in Figure B.6. According to the third property of Def. B.2.6 there will be two statuses: the *norm*, which is set by default, and the *stop* which is set when `acti` is disabled, or terminated, or if some component (`acti` or `source`) has thrown a fault. For applying the style, `source` will be used as operand for `B`, whereas both `source` and `acti` will be operands for `A`. Figure B.7 shows the result of applying the two sequential architectures, whereas Figure B.8 shows the result of applying the Status mngmt in `act` composite.

B.2.6 Architectures of the components in PROC and scope

The `scope` encloses composite components with applied architectures that coordinate their constituent components. We describe the architectures used in each of the `norm`, `evhlrs`, `faulthlrs` and `comphlr`, though we omit the expected safety properties and implementation details.

The `norm` composite encloses a main activity (`act`) and a component that contains event handlers (`evhlrs`). Two coordinating components are attached, namely the `scinit` and the `scfin`, that perform initialization and finalization actions at the beginning (resp. at the end). of `norm`. For example, `scfin` checks whether open IMAs are left after `act` and `evhlrs` have finished. The *Sequential* style is applied, so that `scinit` and `act` are executed sequentially, whereas the *Parallel* style is applied to the *Sequential*'s result and the `evhlrs`. Finally, the *Sequential* style is applied again on the the *Parallel*'s result and `scfin`. The `scinit` and `scfin` are weakly bisimilar to `empty`, thus they are valid operands for these styles. One extra connector is used, that turns off event handlers (evhlrs.*turnOff* port), so that no new events are handled after act has finished (act.*fin*). Note that if norm is inside PROC it will contain the process' starting activity, which in this case must first finish before evhlrs is started. This enables evhlrs to use input received by the starting activity (e.g. for the definition of expected events).

The evhlrs composite encloses a single empty component handles in the trivial case where there are no expected events to handle. In all other cases, it encloses n event receiving components (i.e. a timer or listn) and $m = 2 \times n$ evscope components that handle the incoming events. Each event receiving component is associated with two evscope components that will handle one event occurrence each. For our verification purposes, it is sufficient to consider just two occurrences of each event, thus we use only two evscopes per event. Two evscopes can materialize all the interleavings that are necessary to capture concurrency issues due to parallel handling of the same event. All event receivers are started when evhlrs is started (i.e. *Parallel* style) and they start one of their associated evscopes upon receiving an event. Event receivers can receive subsequent events, until evhlrs is turned off (i.e. all the non-started evscopes are then disabled). Afterwards, evhlrs is considered finished when all evscopes are finished. The evscopes can be concurrently compensated. The behavior of evscope is quite similar to scope, since they share the same structure and connectors. Their difference lies in that evscope treats the event receiver as if it were an enclosed component even though it is not. According to that, evscope will handle any fault thrown by the event receiver and allows the event receiver to write to variables and access the correlation sets of the evscope.

The faulthlrs composite encloses n catch and their associated n act components. Also, a scfin component is attached, as the one used in norm. The *Alternative* style is applied so that only one catch can trigger its associated act when faulthlrs is started. Specifically, upon faulthrs.*startFH* a rendezvous of all catch.*start* ports is invoked in which each catch exports a string that characterizes the faults that it can handle. The rendezvous connector performs a computation which decides on a `catch` that triggers its `act` (i.e. all other `catch` and `act` are then disabled). The fault occurrence is stored as a local variable in `catch`, so that it can be thrown again by `faulthlrs` if a `rethrow` component is executed within `act`. The *Sequential* style is applied so that `scfin` is started after the result of *Alternative* style is finished.

The `comphlr` composite encloses an `act` and a `scfin` coordinated by the *Sequential* style. The `termhlr` composite encloses a single `act`, thus it has no coordination needs.

B.3 Models for basic activities and other atomic components

The models for basic activities and the other atomic components of Table 4.2 are presented here. Note that in the components' figures we omit to include all the `term` ports, in order to keep them uncluttered.

invoke

The `invoke` component performs a service invocation and it has two variants based on whether it invokes a one-way or a request-response operation. Figure B.9 shows the ports included in both variants with solid line and the ports that are specific to each variant with dashed and dotted lines, respectively. The *intern* port is used for the component's internal transitions. The used ports are:

- *read*, to read the partner link and the variables (and correlation sets), for preparing the message.
- *snd_msg*, to send the invocation message;
- *rcv_msg*, when the invocation's response is received
- *write*, to store the message and the retrieved correlation sets.

The component throws possible *correlation violation* (cs_viol), *uninitialized partner role* (unin_role), *uninitialized variable* (unin_var) and *selection failure* (slct_fail) faults during the preparation of the request message. From these faults, the first three can be detected, while the last one is non-deterministically thrown. For the response message, the component throws also the *mismatched assignment* (mis_assg) fault non-deterministically. Also, if the received response corresponds to a fault message (flt_msg) then this fault is thrown by the `invoke`.

reply

The `reply` component (Figure B.10) handles the response of a synchronous operation.

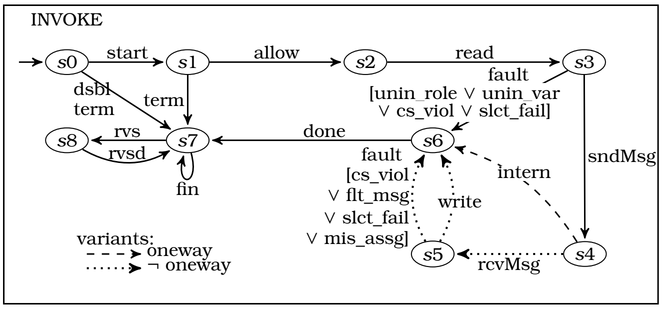


Figure B.9: The `invoke` component with two variants.

The used ports are:

- *chk_ima*, to check that such a request is waiting for response.
- *read*, to read variables and correlation sets for preparing the message
- *close_ima*, to remove request from the waiting list.
- *snd_msg*, to send the response message;

The component detects and throws possible *correlation violation* (cs_viol) and *uninitialized variable* (unin_var) faults. Also, it throws non-deterministically the *mismatched assignment* (mis_assg), *selection failure* (slct_fail), and *missing request* (miss_req) faults.

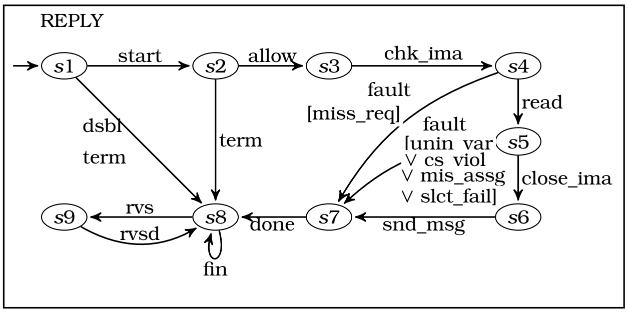


Figure B.10: The `reply` component.

listn

The `listn` component is ommited, since it is similar to the `receive`. Namely, `listn` is different in that: (i) it can receive multiple messages with one open listening endpoint, and (ii) it can be turned off, when it shouldn't receive any more messages.

copy

The copy component (Figure B.11) handles the assignment to a BPEL variable (or a partner link). The component reads the source variable stored in some data component (*read* port) and evaluates its value. If no fault occurs, it assigns the value to the target variable, found in some data component (*write* port). Note that if the copy assigns from a literal value, the *read* port has no effect.

The component detects and throws possible *uninitialized variables* (unin_var), and *uninitialized partner role* (unin_role) faults. Also, it throws the *mismatched assignment* (mis_assg), *selection failure* (slct_fail), and *invalid selection* (inval_slct) faults non-deterministically. These faults may or may not be thrown based on the source and target variables' variant, which can be a variable, a partner link, a literal value or an expression.

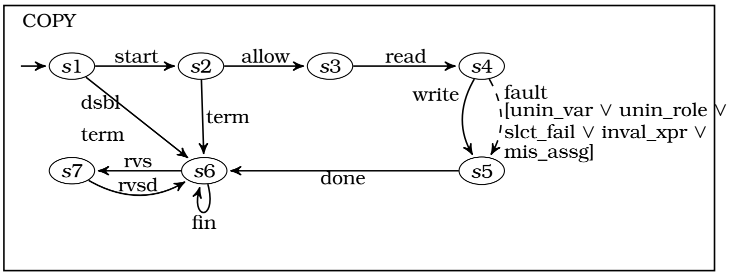


Figure B.11: The copy component with two variants.

compensate

The compensate component (Figure B.12) controls the compensation of one or many scopes. Such a component reside only within the act of the components faulthlrs, comphlr and termhlr. The used ports are:

- *compensate*, to start the compensation of some scope(s).
- *infault*, to receive a fault from the compensated scopes.
- *endrvrs*, to be notified about the end of compensation.
- *termcomp*, to abruptly terminate the compensation.

The component rethrows, through the *fault* port, any fault of the compensated scope. Thus, these faults can be handled by the `compensate`'s enclosing scope.

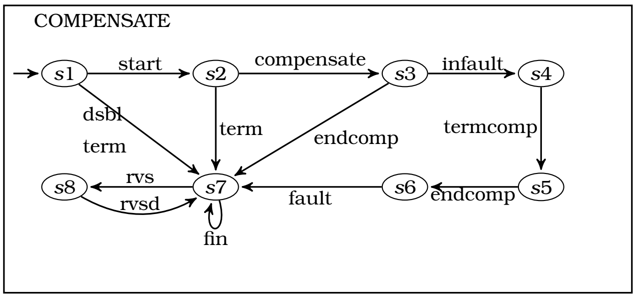


Figure B.12: The `compensate` component.

throw , rethrow

The `throw` component (Figure B.13) is used to throw an explicit internal fault, through the *fault* port. On the other hand, the `rethrow` component (Figure B.14) has a slightly different role; It is placed within `faulthlrs` in order to to rethrow the fault that was originally caught. For this reason, the `rethrow` component uses the *rethrow* port that has different semantics from the *throw* port and it is handled differently by the `scope`'s glue: initially the transferred fault is unknown and it is assigned at the scope level, after it is retrieved by the scope's controller which stores the caught faults.

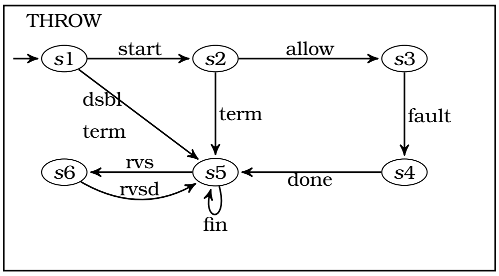


Figure B.13: The `throw` component.

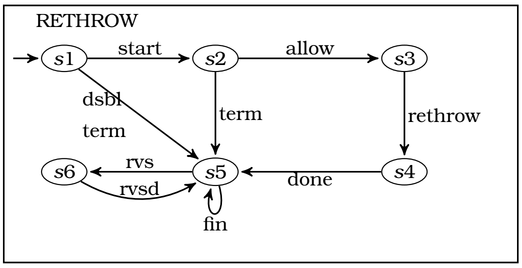


Figure B.14: The `rethrow` component.

exit

The `exit` component (Figure B.15) fires the *exit* port, that causes the whole process's interruption.

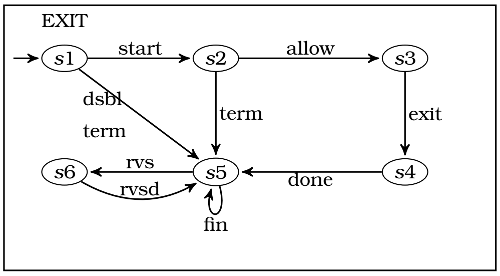


Figure B.15: The `exit` component.

valid

The `valid` component (Figure B.16) validates a variable against its data type (XML schema). Since our modelling assumes only symbolic values for variables, the component cannot detect the *invalid variables* (inval_var) fault, thus the fault is thrown non-deterministically.

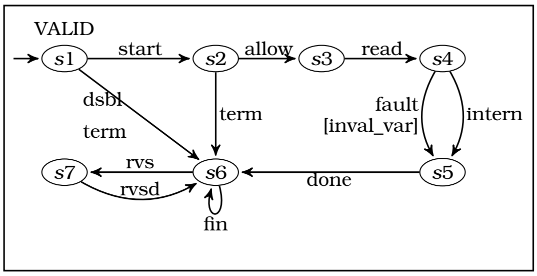


Figure B.16: The `valid` component.

timer

The `timer` component (Figure B.17) is used to model the BPEL activities *onAlarm* and *wait* that handle the firing of a time-out. The time-out is specified using either a duration (relative timestamp) or a date (absolute timestamp). Also, the time-out for *onAlarm* may be periodic, if a time period is given. All the `timer` components of the model are synchronized (*tick* port) so that they update their remaining time accordingly: upon each *tick*, one `timer` expires (whichever is closer to expire), while this `timer's` remaining time is reduced by the other `timer`s remaining times. The used ports are:

- *read*, to read the variables for evaluating the timer's expressions.

- *tick*, to synchronize with other `timers`.
- *trigger*, when there is a time-out.
- *tOff*, when it is turned off.

The component can detect a possible *uninitialized variable* (unin_var) fault, while it throws the *invalid expression value* (inval_xpr) fault non-deterministically.

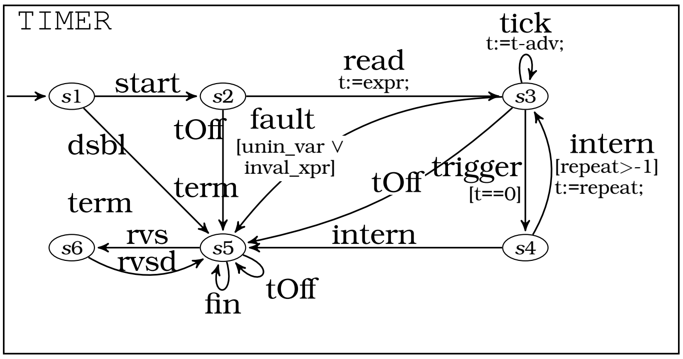


Figure B.17: The `timer` component.

loopctrl

The `loopctrl` component for the *while* and *repeatUntil* activities are shown in Figure B.18 and Figure B.19 respectively. Both components have the same interface, though their behaviors differ, based on whether the first execution depends on a decision or not. The used ports are:

- *read*, to read the variables for evaluating the conditions.
- *trigger*, to start the loop body activity;
- *done_in*, when the loop body activity is finished.
- *tOff*, to exit the loop.

The component can detect a possible *uninitialized variable* (unin_var) fault, while it throws the *invalid expression value* (inval_xpr) fault non-deterministically.

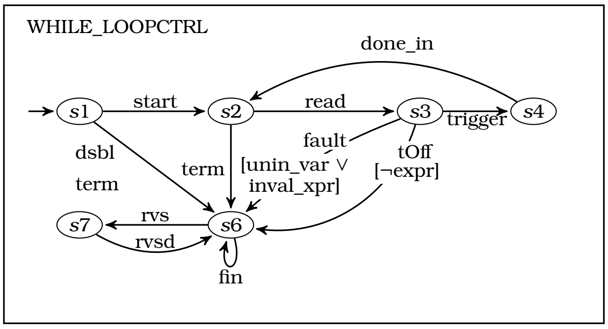


Figure B.18: The `loopctrl` component for the `while` activity.

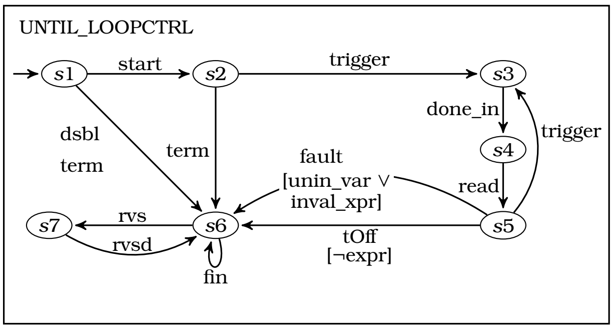


Figure B.19: The `loopctrl` component for the `repeatUntil` activity.

The `loopctrl` for the *forEach* and the parallel *forEach* activities are shown in Figure B.20 and Figure B.21 respectively. The components have the same interface, though their behaviors differ slightly. The used ports are:

- *read*, to read the variables for evaluating the expressions.
- *trigger*, to start the loop body scope;
- *succ*, when the loop body scope is successfully finished.
- *fail*, when the loop body scope is finished but not successfully.
- *tOff*, to exit the loop.

The component can detect a possible *uninitialized variable* (unin_var) fault, while it throws the *invalid expression value* (inval_xpr) fault non-deterministically.

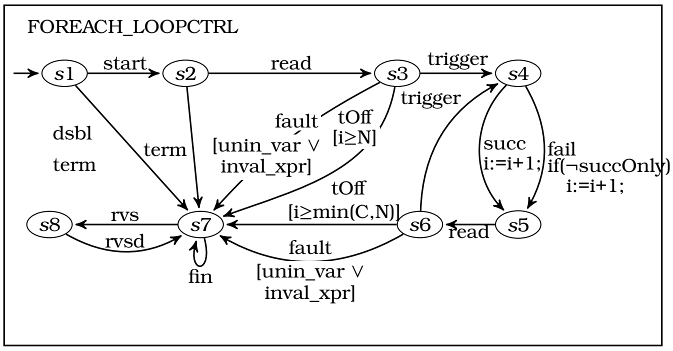


Figure B.20: The `loopctrl` component for the `forEach` activity.

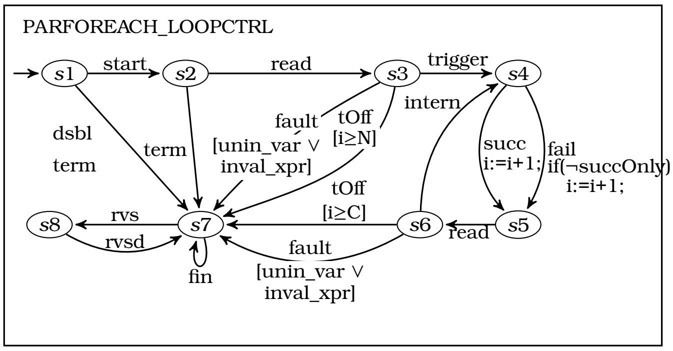


Figure B.21: The `loopctrl` component for the parallel `forEach` activity.

condctrl

The `condctrl` component (Figure B.22) handles a decision for the execution of one out of N components. The used ports are:

- *read$_i$*, to read the variables for evaluating expression i.
- *trigger$_i$*, to start component i;
- *tOff$_i$*, to disable component i.

The component can detect a possible *uninitialized variable* (unin_var) fault, while it throws the *invalid expression value* (inval_xpr) fault non-deterministically.

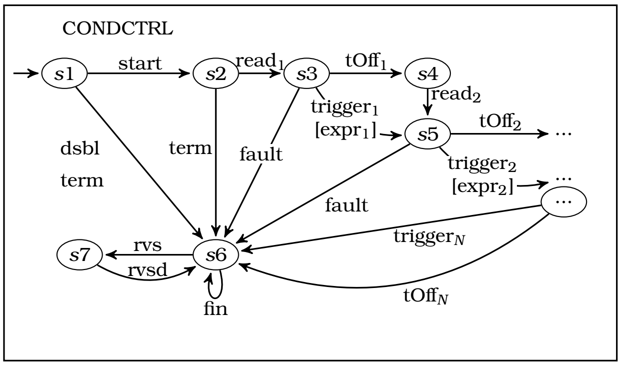


Figure B.22: The `condctrl` component.

B.4 Translation times for test programs

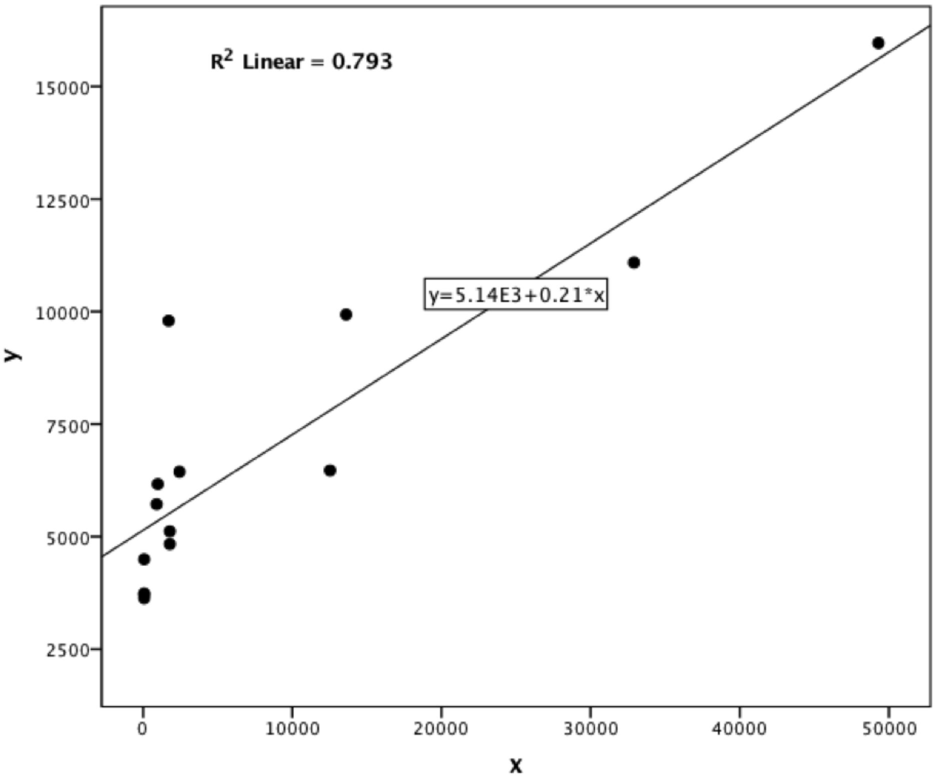


Figure B.23: Regression analysis of translation times for test programs (*x* axis is the number of states, *y* axis in *ms*)

Appendix C

C.1 Language for Contiki REST application design definition (DSL)

This Appendix refers to the DSL syntax and BIP templates, for client actions, which are not mentioned in Section 5.4.3. Moreover, a server process in DSL and its Contiki template are presented.

```
<while boolExp="bool-expr">
    action+
</while>

<if boolExp="bool-expr">
    action+
    <elseif boolExp="bool-expr">*
        action+
    </elseif>
    <else>?
        action+
    </else>
</if>

<wait>
    <onEv evType="event-type"?
            cond=""? >*
        action+
    </onEv>
</wait>
```

Listing C.1: DSL syntax for structured actions

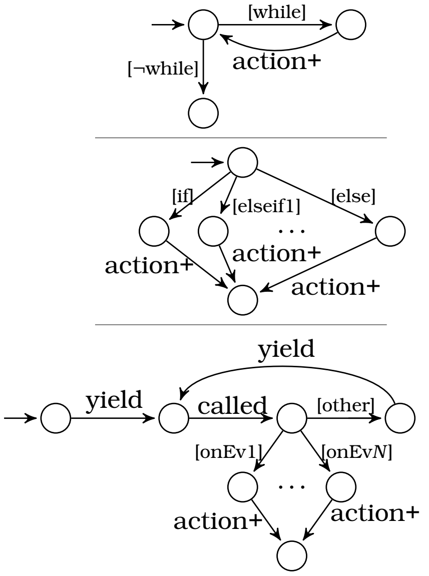


Figure C.1: BIP templates for actions in Listing C.1

The actions can encode control flow structures with nested actions, such loops, conditionals or event handlers. We call these actions *structured* and their syntax is defined in Listing C.1. Moreover, Figure C.1 shows the corresponding BIP template for each of them.

A set of *basic* actions including timeout, event dispatching, message communication, block of code etc. have been also encoded, whose syntax is presented

```
<timeout timer="QName"
    command="set|rset|rstrt|stop"
    (var="QName"|val="unsign-int")? />

<postEv mode="syn|asyn"
    evType="event-type"
    process="QName"?
    var="QName"? />

<sndReq server="QName"
    resource="QName"
    params="QName-list"?
    method="put|get|post|del"
    (contentType="txt|json|xml"
        var="QName")? />

<getResp/>

<exit />

<code> <!--C/C++ code--> </code>
```

Listing C.2: DSL syntax for basic actions

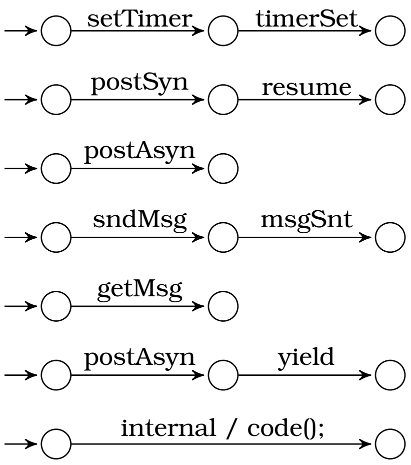


Figure C.2: BIP templates for actions in Listing C.2

in Listing C.2. The BIP behaviour that is instantiated for these actions is shown in Figure C.2, where each action is modelled by the activation of one or two successive ports. For a <timeout>, the process first sets a timer (*timerSet* port) and waits until the timer is set by Contiki (*timerSet* port). The <postEv> action corresponds either to a synchronous or an asynchronous posting. After synchronous posting (*postSyn* port), the process is blocked until be allowed to resume (*resumr* port) by the Contiki. This is in contrast with the asynchronous posting, which does not block the process. With the <sndReq> action, the process creates a message to be sent by Contiki (*sndMsg* port) and it is notified when the message is sent (*msgSnt* port). The <getResp/> action (*getMsg* port) is used when receiving a response message, whereas with the <yield/> action (*yield* port) the process yields. With an <exit/> action, the process posts asynchronously an EXIT event for itself and yields. The process is called, when the EXIT event is scheduled to occur and the exit handler is then triggered. Except from the aforementioned actions, the DSL provides also the <code> element that allows for custom actions which are specified using C/C++ code. In BIP, this element is represented by an internal action (no port is activated) that calls the function with the specified code.

A server process description (Listing C.3) includes a set of (periodic or aperiodic) resource handlers, with handlers for their supporting COAP methods.

The “autoStart” value[1] defines whether the process will be initiated by default, or upon an event by another process. Listing C.4 shows the Contiki syntax of a server process. At the bottom of the template (lines 13-19), there is the definition of a process, which lives only to start a REST engine and activate the resources. The REST engine (whose code is included in the server’s source file) is a predefined Contiki process that implements a REST server, i.e. it invokes the server’s resource handlers either periodically, or upon an incoming request. The communicated messages are passed between the REST engine and the handlers with the *req* and *resp* variables. The template includes two resources, one aperiodic (line 2) and one periodic (line 6). Resource handlers are implemented as a `[resource_id]_handler` function.

```
<server id="..."
    autoStart="[true|false]"? >+
     <resource_handler
            resource_id="..." >+
        <method id="[get|post|put|
            del]{1,4}"/>+
        <code> <!--C/C++ code--> <
            /code>
        <periodic period="int" >
            <!--C/C++ code-->
        </periodic>?
     </resource_handler>
</server>
```

Listing C.3: DSL syntax for a REST server

```
 1 RESOURCE(res1,/*methods*/,...);
 2 void res1_handler(REQUEST* req,
 3                 RESPONSE* resp)
 4 { .... }
 5
 6 PERIODIC_RESOURCE(res2,/*period*/,...);
 7 ... /* res2 handler */
 8 void res2_periodic_handler(REQUEST* req,
 9                 RESPONSE* resp)
10 { .... }
11 /* other resources and handlers */
12
13 PROCESS(server, ...);
14 AUTOSTART_PROCESSES(&server);
15 PROCESS_THREAD(server, ... ){
16   PROCESS_BEGIN();
17   .... /* start rest engine process */
18   .... /* activate resources */
19   PROCESS_END();}
```

Listing C.4: Contiki code template for a REST server

C.2 BIP interactions of the RestModule model with the OS model

In <AppModel>, each <RESTModule> interacts with the <OS> component as it is shown in Figure C.3. For simplicity, only the OS interactions with one process are depicted. Every process is modelled as an atomic component with application-specific behaviour.

[1] The “autoStart” attribute is used for both servers and clients, although it was not shown in Section 5.4.3

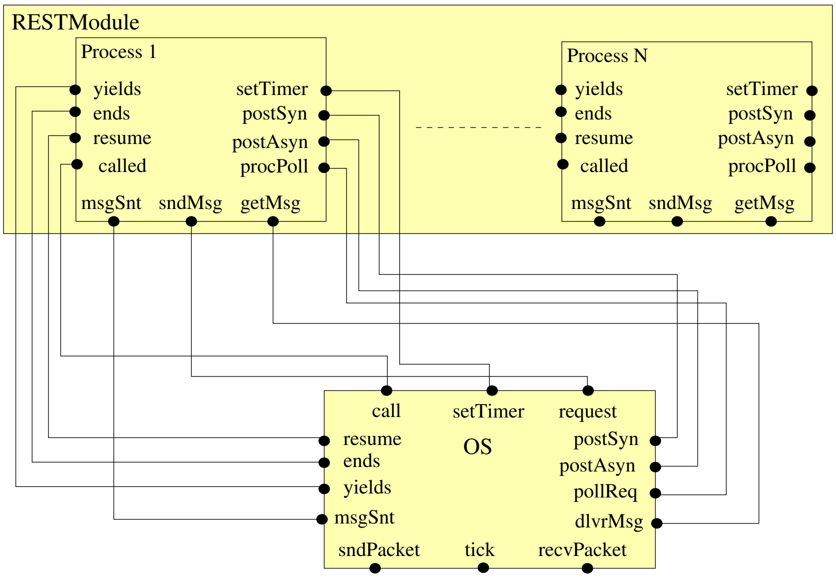


Figure C.3: The RestModule for a Client and its interactions with the OS

A process is called (*called* port), when the OS dispatches an event to it. After the event is handled, the process's execution yields (*yield* port). Posted synchronous or asynchronous events to other processes are passed through the *postSyn* and *postAsyn* ports. For a synchronous event, the process resumes execution (*resume* port) upon the end of event handling. Each process may request polling for itself or other processes and can set deadlines using timers (*setTimer* port) or send a message (*sndMsg* port). The <ConKernel> acknowledges the completion of setting a timer or transmitting a message (*timerSet*, *msgSnt* ports). When the process execution is finished, the *end* port is enabled. The model details for the invocations of REST resource handlers are discussed in [126]. In current model, the supported resource types are periodic, event and actuator.

C.3 *Network stack model parameters*

The model parameters in Table C.1 can be adjusted through the network configuration XML specification (input in step 2). Some parameters concern with the exponential backoff mechanism of the IEEE 802.15.4 standard or the timeout for a packet receipt. Moreover, there are parameters like *macMinBE, macMaxBE, macMaxCSMABackoffs* and *macMaxFrameRetries* that affect the network throughput and the number of channel collisions. Parameter values depend on the transmission time of one symbol (4 bits), denoted by *symbolPeriod*. This time is computed from equation (5.1).

Model parameter	**Value**
aUnitBackoffPeriod	$20 * \mathit{symbolPeriod}$
CCA duration [1]	$8 * \mathit{symbolPeriod}$
macMaxCSMABackoffs	0-5 (default 4)
macAckWaitDuration	$54 * \mathit{symbolPeriod}$
macMinBE	3
macMaxBE	3-8 (default 5)
aMaxFrameRetries	3
t_{data}	$[152, 1064] * \mathit{symbolPeriod}$
t_{ack}	$136 * \mathit{symbolPeriod}$
aTurnaroundTime	$12 * \mathit{symbolPeriod}$
SIFS [2]	$12 * \mathit{symbolPeriod}$
LIFS [2]	$40 * \mathit{symbolPeriod}$

Table C.1: Parameters of the modelled network stack

[1] Clear Channel Assessment (CCA) is the time needed to access the communication channel

[2] Short Interframe Space (SIFS) is the period required for allowing the MAC layer time to process the data received in the physical layer for short data frames and Long Interframe Space (LIFS) is the respective period for long data frames

Bibliography

[1] Joseph Sifakis. “Rigorous System Design”. In: *Foundations and Trends in Electronic Design Automation* 6.4 (2013), pp. 293–362.

[2] A. Basu, B. Bensalem, M. Bozga, J. Combaz, M. Jaber, T. H. Nguyen, and J. Sifakis. “Rigorous Component-Based System Design Using the BIP Framework”. In: *IEEE Software* 28.3 (2011), pp. 41–48.

[3] Roy Thomas Fielding. “Architectural Styles and the Design of Network-based Software Architectures”. AAI9980887. PhD thesis. 2000. ISBN: 0-599-87118-0.

[4] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. “Contiki-a lightweight and flexible operating system for tiny networked sensors”. In: *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*. IEEE. 2004, pp. 455–462.

[5] Brent Hailpern and Padmanabhan Santhanam. “Software debugging, testing, and verification”. In: *IBM Systems Journal* 41.1 (2002), pp. 4–12.

[6] Gary T Leavens and Murali Sitaraman. *Foundations of component-based systems*. Cambridge University Press, 2000.

[7] Alan W Brown and Kurt C Wallnan. “Engineering of component-based systems”. In: *Engineering of Complex Computer Systems, 1996. Proceedings., Second IEEE International Conference on*. IEEE. 1996, pp. 414–422.

[8] Saddek Bensalem, Andreas Griesmayer, Axel Legay, Thanh-Hung Nguyen, Joseph Sifakis, and Rongjie Yan. “D-Finder 2: towards efficient correctness of incremental design”. In: *Proceedings of the 3^{rd} international conference on NASA Formal methods*. NFM’11. Pasadena, CA: Springer, 2011, pp. 453–458.

[9] Ananda Basu, Saddek Bensalem, Marius Bozga, Paraskevas Bourgos, and Joseph Sifakis. "Rigorous System Design: The BIP Approach". In: *Mathematical and Engineering Methods in Computer Science*. Ed. by Zdeněk Kotásek, Jan Bouda, Ivana Černá, Lukáš Sekanina, Tomáš Vojnar, and David Antoš. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–19.

[10] Anastasia Mavridou, Eduard Baranov, Simon Bliudze, and Joseph Sifakis. "Configuration logics: Modeling architecture styles". In: *Journal of Logical and Algebraic Methods in Programming* 86.1 (2017), pp. 2 –29.

[11] CMMI Product Team. *CMMI for Acquisition, Version 1.2*. Tech. rep. CMU/SEI-2007-TR-017. Software Engineering Institute, Carnegie Mellon University, 2007. URL: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=8451.

[12] Alessandro Cimatti, Marco Roveri, Angelo Susi, and Stefano Tonetta. "Validation of Requirements for Hybrid Systems: a Formal Approach". In: 21 (Nov. 2012).

[13] Ananda Basu, Saddek Bensalem, Marius Bozga, Paraskevas Bourgos, Mayur Maheshwari, and Joseph Sifakis. "Component Assemblies in the Context of Manycore". In: *10th Int. Symp. on Formal Methods for Components and Objects (FMCO 2011)*. Ed. by Bernhard Beckert, Ferruccio Damiani, Frank S. de Boer, and Marcello M. Bonsangue. Vol. 7542. Lecture Notes in Computer Science. 2013, pp. 314–333.

[14] Paul Attie, Eduard Baranov, Simon Bliudze, Mohamad Jaber, and Joseph Sifakis. "A general framework for architecture composability". In: *Formal Aspects of Computing* 28.2 (2016), pp. 207–231.

[15] Anastasia Mavridou, Emmanouela Stachtiari, Simon Bliudze, Anton Ivanov, Panagiotis Katsaros, and Joseph Sifakis. "Architecture-based Design: A Satellite On-board Software Case Study". In: *Proceedings of the 13th International Conderence of Formal Aspects in Component Software*. 2016.

[16] Anastasia Mavridou, Eduard Baranov, Simon Bliudze, and Joseph Sifakis. "Architecture Diagrams: A Graphical Language for Architecture

Style Specification”. In: *Proceedings 9th Interaction and Concurrency Experience (ICE)*. Vol. 223. EPTCS. 2016, pp. 83–97.

[17] Dennis M Buede and William D Miller. *The engineering design of systems: models and methods*. John Wiley & Sons, 2016.

[18] Ariel Fuxman, Lin Liu, John Mylopoulos, Marco Pistore, Marco Roveri, and Paolo Traverso. “Specifying and analyzing early requirements in Tropos”. In: *Requirements Engineering* 9.2 (2004), pp. 132–150.

[19] Elizabeth Hull, Ken Jackson, and Jeremy Dick. *Requirements Engineering*. 3rd. New York, NY, USA: Springer, 2010.

[20] Albert Benveniste, Benoît Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto Sangiovanni-Vincentelli, Werner Damm, Tom Henzinger, and Kim Guldstrand Larsen. *Contracts for Systems Design: Theory*. Research Report RR-8759. Inria Rennes Bretagne Atlantique ; INRIA, July 2015, p. 86. URL: https://hal.inria.fr/hal-01178467.

[21] Allan Berrocal Rojas and Gabriela Barrantes Sliesarieva. “Automated Detection of Language Issues Affecting Accuracy, Ambiguity and Verifiability in Software Requirements Written in Natural Language”. In: *Proceedings of the NAACL HLT 2010 Young Investigators Workshop on Computational Approaches to Languages of the Americas*. YIWCALA ’10. 2010.

[22] Deborah Anne Baker. “The Use of Requirements in Rigorous System Design”. PhD thesis. University of Southern California, 1982.

[23] Steven P. Miller, Alan C. Tribble, Michael W. Whalen, and Mats P. E. Heimdahl. “Proving the Shalls: Early Validation of Requirements Through Formal Methods”. In: *International Journal on Software Tools for Technology Transfer* 8.4-5 (2006), pp. 303–319.

[24] Pamela Zave and Michael Jackson. “Four dark corners of requirements engineering”. In: *ACM transactions on Software Engineering and Methodology (TOSEM)* 6.1 (1997), pp. 1–30.

[25] Ajitha Rajan and Thomas Wahl, eds. *CESAR - Cost-efficient Methods and Processes for Safety-relevant Embedded Systems*. Springer Vienna,

2013. ISBN: 978-3-7091-1386-8. URL: http://www.springer.com/engineering/production+engineering/book/978-3-7091-1386-8 (visited on 04/26/2013).

[26] Alistair Mavin and Philip Wilkinson. “Big Ears (The Return of "Easy Approach to Requirements Engineering")”. In: *RE 2010, 18th IEEE International Requirements Engineering Conference*. 2010, pp. 277–282.

[27] Alistair Mavin, Philip Wilkinson, Adrian Harwood, and Mark Novak. “Easy Approach to Requirements Syntax (EARS)”. In: *Proceedings of the 2009 17th IEEE International Requirements Engineering Conference, RE*. RE ’09. IEEE, 2009, pp. 317–322.

[28] Anastasia Mavridou, Emmanouela Stachtiari, Simon Bliudze, Anton Ivanov, Panagiotis Katsaros, and Joseph Sifakis. *Architecture-based Design: A Satellite On-Board Software Case Study*. Tech. rep. 221156. https://infoscience.epfl.ch/record/221156. EPFL, Sept. 2016.

[29] ECSS-E-ST-10C Working Group. “ECSS-E-ST-10C - System engineering general requirements”. In: *European Cooperation for Space Standardization (ECSS), ESA Publications* (2009), pp. 1–100.

[30] Mike Mannion, Barry Keepence, and David Harper. “Using viewpoints to define domain requirements”. In: *IEEE software* 15.1 (1998), pp. 95–102.

[31] Michael Jackson. “Problem analysis and structure”. In: *Engineering Theories of Software Construction (Proceedings of the NATO Summer School*. IOS Press, 2000.

[32] ECSS-M-ST-10C Working Group. “ECSS-M-ST-10C - Space project management: Project planning and implementation”. In: *European Cooperation for Space Standardization (ECSS) , ESA Publications* (2009), pp. 1–50.

[33] Nesredin Mahmud, Cristina Seceleanu, and Oscar Ljungkrantz. “Specification and Semantic Analysis of Embedded Systems Requirements: From Description Logic to Temporal Logic”. In: *Software Engineering and Formal Methods: 15th International Conference, SEFM 2017, Trento, Italy, September 4-8, 2017, Proceedings*. Ed. by Alessandro Cimatti and Marjan Sirjani. Vol. 10469. Lecture Notes in Computer Science. 2017, pp. 332–348.

[34] *BIP tools.* http://www-verimag.imag.fr/BIP-Tools,93.html.

[35] Anastasia Mavridou, Joseph Sifakis, and Janos Sztipanovits. "DesignBIP: A Design Studio for Modeling and Generating Systems with BIP". In: *Proceedings of the 1st International Workshop on Methods and Tools for Rigorous System Design (MetRID 2018).* 2018.

[36] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. "The nuXmv Symbolic Model Checker". In: *Computer Aided Verification.* Ed. by Armin Biere and Roderick Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 334–342.

[37] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. "Synchronous Observers and the Verification of Reactive Systems". In: *Proceedings of the Third International Conference on Methodology and Software Technology: Algebraic Methodology and Software Technology.* AMAST '93. Springer, 1994, pp. 83–96.

[38] Andreas Mitschke et al. *Requirements Specification Language and Requirements Meta Model.* Tech. rep. D_SP2_R2.1_M1. CESAR - Cost efficient methods and processes for safety relevant embedded systems, 2010.

[39] Odd Ivar Lindland, Guttorm Sindre, and Arne Sølvberg. "Understanding Quality in Conceptual Modeling". In: *IEEE Software* 11.2 (Mar. 1994), pp. 42–49.

[40] Felix Leung and Narasimha Bolloju. "Analyzing the quality of domain models developed by novice systems analysts". In: *System Sciences, 2005. HICSS'05. Proceedings of the 38th Annual Hawaii International Conference on.* IEEE. 2005, 188b–188b.

[41] NXP. *UM10204: I2C-bus specification and user manual.* Standard. June 2007.

[42] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series).* The MIT Press, 2008. ISBN: 026202649X, 9780262026499.

[43] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. “Patterns in Property Specifications for Finite-state Verification”. In: *Proceedings of the 21st International Conference on Software Engineering*. ICSE ’99. Los Angeles, California, USA: ACM, 1999, pp. 411–420.

[44] Simon Bliudze, Alessandro Cimatti, Mohamad Jaber, Sergio Mover, Marco Roveri, Wajeb Saab, and Qiang Wang. “Formal verification of infinite-state BIP models”. In: *13th International Symposium on Automated Technology for Verification and Analysis (ATVA ’15)*. Vol. 9364. LNCS. Nov. 2015, pp. 326–343. DOI: 10.1007/978-3-319-24953-7_25.

[45] Stefano Rossi, Anton Ivanov, Gael Soudan, Volker Gass, Christine Hollenstein, and Markus Rothacher. “CubETH magnetotorquers: Design and tests for a CubeSat mission”. In: vol. 153. 2015, pp. 1513–1530.

[46] Marco Bozzano, Alessandro Cimatti, Joost-Pieter Katoen, Panagiotis Katsaros, Konstantinos Mokos, Viet Yen Nguyen, Thomas Noll, Bart Postma, and Marco Roveri. “Spacecraft early design validation using formal methods”. In: *Reliability Engineering & System Safety* 132 (2014), pp. 20 –35.

[47] Michael W Whalen, Andrew Gacek, Darren Cofer, Anitha Murugesan, Mats PE Heimdahl, and Sanjai Rayadurgam. “Your" What" Is My" How": Iteration and Hierarchy in System Design”. In: *IEEE software* 30.2 (2013), pp. 54–60.

[48] Sanjai Rayadurgam, John Komp, Lian Duan, Baek-Gyu Kim, Oleg Sokolsky, and Insup Lee. “From Requirements to Code: Model Based Development of A Medical Cyber Physical System”. In: 9062 (2017), p. 96.

[49] Anitha Murugesan, Michael W Whalen, Sanjai Rayadurgam, and Mats PE Heimdahl. “Compositional verification of a medical device system”. In: *ACM SIGAda Ada Letters* 33.3 (Nov. 2013), pp. 51–64.

[50] Marco Bozzano, Alessandro Cimatti, Anthony Fernandes Pires, D. Jones, G. Kimberly, T. Petri, R. Robinson, and Stefano Tonetta. “Formal Design and Safety Analysis of AIR6110 Wheel Brake System”. In: *International Conference on Computer Aided Verification*. Ed. by Daniel Kroening and Corina S. Păsăreanu. Springer. 2015, pp. 518–535.

[51] Michel D. Ingham, John Day, Kenneth Donahue, Alexander Kadesch, Andrew Kennedy, Mohammed Omair Khan, Ethan Post, and Shaun Standley. "A Model-based Approach to Engineering Behavior of Complex Aerospace Systems". In: *Infotech@Aerospace 2012, Garden Grove, California, USA, June 19-21, 2012.* 2012.

[52] Alessandro Cimatti, Michele Dorigatti, and Stefano Tonetta. "OCRA: A tool for checking the refinement of temporal contracts". In: *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on.* IEEE. 2013, pp. 702–705.

[53] Jamieson M. Cobleigh, George S. Avrunin, and Lori A. Clarke. "Breaking Up is Hard to Do: An Evaluation of Automated Assume-guarantee Reasoning". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 17.2 (May 2008).

[54] Martin Böschen, Ralf Bogusch, Anabel Fraga, and Christian Rudat. "Bridging the Gap between Natural Language Requirements and Formal Specifications". In: *Joint Proceedings of REFSQ-2016 Workshops, Doctoral Symposium, Research Method Track, and Poster Track co-located with the 22nd International Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2016), Gothenburg, Sweden, March 14, 2016.* 2016.

[55] Jinxin Lin, Mark S. Fox, and Taner Bilgic. "A Requirement Ontology for Engineering Design". In: *Concurrent Engineering* 4.3 (1996), pp. 279–291.

[56] David A Wagner, Matthew B Bennett, Robert Karban, Nicolas Rouquette, Steven Jenkins, and Michel Ingham. "An ontology for State Analysis: Formalizing the mapping to SysML". In: *Aerospace Conference, 2012 IEEE.* IEEE. 2012, pp. 1–16.

[57] Werner Damm, Hardi Hungar, Bernhard Josko, Thomas Peikenkamp, and Ingo Stierand. "Using contract-based component specifications for virtual integration testing and architecture design." In: *DATE.* IEEE, 2011, pp. 1023–1028.

[58] Stefan Farfeleder, Thomas Moser, Andreas Krall, Tor Stalhane, Herbert Zojer, and Christian Panis. “DODT: Increasing requirements formalism using domain ontologies for improved embedded systems development.” In: *DDECS*. Ed. by Rolf Kraemer, Adam Pawlak, Andreas Steininger, Mario Schölzel, Jaan Raik, and Heinrich Theodor Vierhaus. IEEE, 2011, pp. 271–274.

[59] Haruhiko Kaiya and Motoshi Saeki. “Ontology Based Requirements Analysis: Lightweight Semantic Processing Approach”. In: *Proceedings of the Fifth International Conference on Quality Software*. QSIC ’05. IEEE, 2005, pp. 223–230.

[60] Levi Lúcio, Salman Rahman, Chih-Hong Cheng, and Alistair Mavin. “Just Formal Enough? Automated Analysis of EARS Requirements”. In: *NASA Formal Methods Symposium*. 2017, pp. 427–434.

[61] Chih-Hong Cheng, Yassine Hamza, and Harald Ruess. “Structural Synthesis for GXW Specifications”. In: *International Conference on Computer Aided Verification*. 2016, pp. 95–117.

[62] *Unified Modeling Language Specification, Version 2.5.1*. http://www.omg.org/spec/UML/2.5.1/.

[63] Nenad Medvidovic and Richard N Taylor. “A classification and comparison framework for software architecture description languages”. In: *IEEE Transactions on software engineering* 26.1 (2000), pp. 70–93.

[64] Eoin Woods and Rich Hilliard. “Architecture description languages in practice session report”. In: *Software Architecture, 2005. WICSA 2005. 5th Working IEEE/IFIP Conference on*. IEEE. 2005, pp. 243–246.

[65] Mourad Oussalah, Adel Smeda, and Tahar Khammaci. “An explicit definition of connectors for component-based software architecture”. In: *Engineering of Computer-Based Systems, 2004. Proceedings. 11th IEEE International Conference and Workshop on the*. IEEE. 2004, pp. 44–51.

[66] Rob Van Ommering, Frank Van Der Linden, Jeff Kramer, and Jeff Magee. “The Koala component model for consumer electronics software”. In: *Computer* 33.3 (2000), pp. 78–85.

[67] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, and Antony Tang. “What industry needs from architectural languages: A survey”. In: *IEEE Transactions on Software Engineering* 39.6 (2013), pp. 869–891.

[68] Robert Allen and David Garlan. “A formal basis for architectural connection”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 6.3 (1997), pp. 213–249.

[69] Sol Greenspan, John Mylopoulos, and Alex Borgida. “On Formal Requirements Modeling Languages: RML Revisited”. In: *Proceedings of the 16th International Conference on Software Engineering*. ICSE ’94. Sorrento, Italy: IEEE, 1994, pp. 135–147.

[70] Liana Barachisio Lisboa, Vinicius Cardoso Garcia, Eduardo Santana de Almeida, and Silvio Romero Meira de Lemos. “ToolDAy: A Tool for Domain Analysis”. In: *Int. J. Softw. Tools Technol. Transf.* 13.4 (Aug. 2011), pp. 337–353.

[71] Azadeh Alebrahim, Maritta Heisel, and Rene Meis. “A structured approach for eliciting, modeling, and using quality-related domain knowledge”. In: *International Conference on Computational Science and Its Applications*. Springer. 2014, pp. 370–386.

[72] Nesredin Mahmud, Cristina Seceleanu, and Oscar Ljungkrantz. “ReSA tool: Structured requirements specification and SAT-based consistency-checking”. In: *Computer Science and Information Systems (FedCSIS), 2016 Federated Conference on*. IEEE. 2016, pp. 1737–1746.

[73] Philipp Reinkemeier, Ingo Stierand, Philip Rehkop, and Stefan Henkler. “A pattern-based requirement specification language: Mapping automotive specific timing requirements”. In: *Software Engineering 2011 - Workshopband (inkl. Doktorandensymposium), Fachtagung des GI-Fachbereichs Softwaretechnik, 21.-25.02.2011, Karlsruhe*. 2011, pp. 99–108.

[74] Lars Grunske. “Specification Patterns for Probabilistic Quality Properties”. In: *Proceedings of the 30th International Conference on Software Engineering*. ICSE ’08. Leipzig, Germany: ACM, 2008, pp. 31–40.

[75] Ayoub Nouri, Saddek Bensalem, Marius Bozga, Benoit Delahaye, Cyrille Jegourel, and Axel Legay. "Statistical Model Checking QoS Properties of Systems with SBIP". In: *Int. J. Softw. Tools Technol. Transf.* 17.2 (Apr. 2015), pp. 171–185.

[76] Tesnim Abdellatif, Jacques Combaz, and Joseph Sifakis. "Rigorous implementation of real-time systems - from theory to application". In: *Mathematical Structures in Computer Science* 23.4 (2013), pp. 882–914.

[77] Souha Ben Rayana, Marius Bozga, Saddek Bensalem, and Jacques Combaz. "RTD-Finder: A Tool for Compositional Verification of Real-Time Component-Based Systems". In: *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. 2016, pp. 394–406.

[78] William Swartout and Robert Balzer. "On the Inevitable Intertwining of Specification and Implementation". In: *Commun. ACM* 25.7 (July 1982), pp. 438–440. ISSN: 0001-0782.

[79] Bashar Nuseibeh. "Weaving Together Requirements and Architectures". In: *Computer* 34.3 (Mar. 2001), pp. 115–117.

[80] Paulina Paraponiari and George Rahonis. "On weighted configuration logics". In: *CoRR* abs/1704.04969 (2017).

[81] Chang-ai Sun, Yan Zhao, Lin Pan, Huai Liu, and Tsong Yueh Chen. "Automated testing of WS-BPEL service compositions: A scenario-oriented approach". In: *IEEE Transactions on Services Computing* 11.4 (2018), pp. 616–629.

[82] Maurice H Beek, Antonio Bucchiarone, and Stefania Gnesi. "Formal Methods for Service Composition". In: *Annals of Mathematics, Computing and Teleinformatics* 1.5 (2007), pp. 1–14.

[83] Simon Bliudze and Joseph Sifakis. "A Notion of Glue Expressiveness for Component-Based Systems". In: *CONCUR 2008 - Concurrency Theory*. Vol. 5201. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008, pp. 508–522.

[84] Emmanouela Stachtiari, Anakreon Mentis, and Panagiotis Katsaros. "Rigorous analysis of service composability by embedding WS-BPEL into the BIP component framework". In: *Web Services (ICWS), 2012 IEEE 19th International Conference on.* IEEE. 2012, pp. 319–326.

[85] *BPEL process modelling tools.* http://depend.csd.auth.gr/research/BpelProcessModelling. 2017.

[86] Seema Jehan, Ingo Pill, and Franz Wotawa. "BPEL Integration Testing". In: *International Conference on Fundamental Approaches to Software Engineering.* Springer. 2015, pp. 69–83.

[87] Najah Ben Said and Takoua Abdellatif. "A Robust Framework for Securing Composed Web Services". In: *Formal Aspects of Component Software: 12th International Conference, FACS 2015, Niterói, Brazil, October 14-16, 2015, Revised Selected Papers.* Vol. 9539. 2016, p. 105.

[88] Emmanouela Stachtiari, Nikos Vesyropoulos, George Kourouleas, Christos K Georgiadis, and Panagiotis Katsaros. "Correct-by-construction web service architecture". In: *Service Oriented System Engineering (SOSE), 2014 IEEE 8th International Symposium on.* IEEE. 2014, pp. 47–58.

[89] Saddek Bensalem, Marius Bozga, Benoit Delahaye, Cyrille Jegourel, Axel Legay, and Ayoub Nouri. "Statistical Model Checking Qos Properties of Systems with SBIP". In: *Proceedings of the 5th International Conference on Leveraging Applications of Formal Methods, Verification and Validation: Technologies for Mastering Change - Volume Part I.* ISoLA'12. Heraklion, Crete, Greece, 2012, pp. 327–341. ISBN: 978-3-642-34025-3.

[90] Kai C Wong and W Murray Wonham. "Hierarchical control of discrete-event systems". In: *Discrete Event Dynamic Systems* 6.3 (1996), pp. 241–273.

[91] Rob J. van Glabbeek and W. Peter Weijland. "Branching Time and Abstraction in Bisimulation Semantics". In: *J. ACM* 43.3 (May 1996), pp. 555–600.

[92] OASIS. *Web Services Business Process Execution Language Version 2.0.* OASIS, 2007. URL: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[93] Fabrizio Montesi and Marco Carbone. “Programming Services with Correlation Sets”. In: *9th International Conference on Service-Oriented Computing (ICSOC), Paphos, Cyprus, December 5-8, 2011 Proceedings*. Springer, 2011, pp. 125–141.

[94] Chun Ouyang, Eric Verbeek, Wil M.P. van der Aalst, Stephan Breutel, Marlon Dumas, and Arthur H.M. ter Hofstede. “Formal semantics and analysis of control flow in WS-BPEL”. In: *Science of Computer Programming* 67.2 (2007), pp. 162 –198.

[95] Pavel Parizek and Jiri Adamek. “Checking session-oriented interactions between web services”. In: *Software Engineering and Advanced Applications, 2008. SEAA’08. 34th Euromicro Conference*. IEEE. 2008, pp. 3–10.

[96] Paul Hudak. “Modular domain specific languages and tools”. In: *Software Reuse, 1998. Proceedings. Fifth International Conference on*. IEEE. 1998, pp. 134–142.

[97] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986. ISBN: 0-201-10088-6.

[98] Matjaz. B. Juric. *A Hands-on Introduction to BPEL*. http://www.oracle.com/technetwork/articles/matjaz-bpel1-090575.html. 2017.

[99] Matjaz. B. Juric. *A Hands-on Introduction to BPEL, Part 2: Advanced BPEL*. http://www.oracle.com/technetwork/articles/matjaz-bpel2-082861.html. 2017.

[100] Niels Lohmann. “A feature-complete Petri net semantics for WS-BPEL 2.0”. In: *International Workshop on Web Services and Formal Methods*. Springer. 2007, pp. 77–91.

[101] Niels Lohmann, Eric Verbeek, Chun Ouyang, and Christian Stahl. “Comparing and evaluating Petri net semantics for BPEL”. In: *International Journal of Business Process Integration and Management* 4.1 (2009), pp. 60–73.

[102] Wil M. P. van der Aalst, Marlon Dumas, Chun Ouyang, Anne Rozinat, and Eric Verbeek. “Conformance Checking of Service Behavior”. In: *ACM Trans. Internet Technol.* 8.3 (May 2008), 13:1–13:30.

[103] Idir Aït-Sadoune and Yamine Aït Ameur. “Stepwise Development of Formal Models for Web Services Compositions: Modelling and Property Verification”. In: *Trans. Large-Scale Data- and Knowledge-Centered Systems* 10 (2013), pp. 1–33.

[104] Barry Norton, Simon Foster, and Andrew Hughes. “A Compositional Operational Semantics for OWL-S”. In: *Proceedings of the 2005 International Conference on European Performance Engineering, and Web Services and Formal Methods, International Conference on Formal Techniques for Computer Systems and Business Processes*. EPEW’05/WS-FM’05. Springer, 2005, pp. 303–317.

[105] Elie Fares, Jean-Paul Bodeveix, and Mamoun Filali. “Design of a BPEL verification tool”. In: *International Workshop on Web Services and Formal Methods*. Springer. 2011, pp. 95–110.

[106] Bernard Berthomieu, Jean-Paul Bodeveix, Mamoun Filali, Hubert Garavel, Frédéric Lang, Florent Peres, Rodrigo Saad, Jan Stoecker, François Vernadat, P Gaufillet, et al. “The syntax and semantics of Fiacre”. In: *Repport LAAS* 07264 (2007).

[107] Gwen Salaun, Lucas Bordeaux, and Marco Schaerf. “Describing and reasoning on web services using process algebra”. In: *Proceedings. IEEE International Conference on Web Services, 2004*. IEEE. 2004, pp. 43–50.

[108] Roberto Lucchi and Manuel Mazzara. “A pi-calculus based semantics for WS-BPEL”. In: *Journal of Logic and Algebraic Programming* 70.1 (2007), pp. 96–118.

[109] Andrea Ferrara. “Web services: a process algebra approach”. In: *Proceedings of the 2nd international conference on Service oriented computing*. ACM. 2004, pp. 242–251.

[110] Samira Tasharofi, Mohsen Vakilian, Roshanak Zilouchian Moghaddam, and Marjan Sirjani. “Modeling web service interactions using the coordination language Reo”. In: *International Workshop on Web Services and Formal Methods*. Springer. 2007, pp. 108–123.

[111] Emmanuel Baccelli, Oliver Hahm, Mesut Gunes, Matthias Wahlisch, and Thomas C Schmidt. “RIOT OS: Towards an OS for the Internet of Things”. In: *2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE. 2013, pp. 79–80.

[112] Lars Schor, Philipp Sommer, and Roger Wattenhofer. “Towards a zero-configuration wireless sensor network architecture for smart buildings”. In: *BuildSys’09*. ACM. 2009, pp. 31–36.

[113] J. Beal, D. Pianini, and M. Viroli. “Aggregate Programming for the Internet of Things”. In: *Computer* 48.9 (2015), pp. 22–30. ISSN: 0018-9162. DOI: 10.1109/MC.2015.261.

[114] Kirak Hong, David Lillethun, Umakishore Ramachandran, Beate Ottenwälder, and Boris Koldehofe. “Mobile Fog: A Programming Model for Large-scale Applications on the Internet of Things”. In: *Proceedings of the Second ACM SIGCOMM Workshop on Mobile Cloud Computing*. MCC ’13. Hong Kong, China: ACM, 2013.

[115] Stefan Nastic, Sanjin Sehic, Michael Vögler, Hong-Linh Truong, and Schahram Dustdar. “PatRICIA – A Novel Programming Model for IoT Applications on Cloud Platforms”. In: *Proceedings of the 2013 IEEE 6th International Conference on Service-Oriented Computing and Applications*. SOCA ’13. IEEE, 2013.

[116] Ryo Sugihara and Rajesh K. Gupta. “Programming Models for Sensor Networks: A Survey”. In: *ACM TOSN* 4.2 (Apr. 2008), 8:1–8:29. ISSN: 1550-4859.

[117] Angelo P Castellani, Nicola Bui, Paolo Casari, Michele Rossi, Zach Shelby, and Michele Zorzi. “Architecture and protocols for the internet of things: A case study”. In: *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference on*. IEEE. 2010, pp. 678–683.

[118] Deze Zeng, Song Guo, and Zixue Cheng. "The Web of Things: A Survey (Invited Paper)". In: *Journal of Communications* 6.6 (2011), pp. 424–438.

[119] Walter Colitti, Kris Steenhaut, and Niccolò De Caro. "Integrating Wireless Sensor Networks with the Web". In: *IP+SN'11*. 2011.

[120] Zach Shelby, Klaus Hartke, and Carsten Bormann. "The Constrained Application Protocol (CoAP)". In: (2014).

[121] Qing Cao, Tarek Abdelzaher, John Stankovic, Kamin Whitehouse, and Liqian Luo. "Declarative Tracepoints: A Programmable and Application Independent Debugging System for Wireless Sensor Networks". In: *SenSys'08*. Raleigh, NC, USA: ACM, 2008, pp. 85–98. ISBN: 978-1-59593-990-6.

[122] Joakim Eriksson, Fredrik Österlind, Niclas Finne, Nicolas Tsiftes, Adam Dunkels, Thiemo Voigt, Robert Sauter, and Pedro José Marrón. "COOJA/MSPSim: interoperability testing for wireless sensor networks". In: *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). 2009, p. 27.

[123] Ayoub Nouri, Marius Bozga, Anca Molnos, Axel Legay, and Saddek Bensalem. "ASTROLABE: A Rigorous Approach for System-Level Performance Modeling and Analysis". In: *ACM Trans. Embed. Comput. Syst.* 15.2 (Mar. 2016).

[124] Richard E. Schantz, Joseph P. Loyall, Craig Rodrigues, and Douglas C. Schmidt. "Controlling quality-of-service in distributed real-time and embedded systems via adaptive middleware". In: *Software: Practice and Experience* 36.11-12 (2006), pp. 1189–1208.

[125] Francois Despaux. "Modelling and evaluation of the end to end delay in WSN". Theses. Université de Lorraine, Sept. 2015. URL: https://hal.inria.fr/tel-01241044.

[126] Alexios Lekidis, Emmanouela Stachtiari, Panagiotis Katsaros, Marius Bozga, and Christos K Georgiadis. "Using BIP to reinforce correctness of resource-constrained IoT applications". In: *International Symposium on Industrial Embedded Systems (SIES)*. IEEE. 2015, pp. 1–10.

[127] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. “Internet of Things (IoT): A vision, architectural elements, and future directions”. In: *Future Gen. Computer Systems* 29.7 (2013), pp. 1645–1660.

[128] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. “P: Safe Asynchronous Event-driven Programming”. In: *ACM SIGPLAN Notices*. Vol. 48. 6. ACM, 2013, pp. 321–332.

[129] Matthias Kovatsch, Simon Duquennoy, and Adam Dunkels. “A low-power CoAP for Contiki”. In: *MASS’11*. IEEE. 2011, pp. 855–860.

[130] Alexios Lekidis. “Design flow for the rigorous development of networked embedded systems”. Theses. Universite Grenoble Alpes, Dec. 2015. URL: https://tel.archives-ouvertes.fr/tel-01261936.

[131] Dalen Abraham, Mohammad Shabbir Alam, Jean-Pierre Duplessis, Trevor W Freeman, Bill Hanlon, Anton W Krantz, Scott Manchester, and Benjamin Nick. *XML schema for network device configuration*. US Patent 7,657,612. 2010.

[132] Gang Zhou, Tian He, Sudha Krishnamurthy, and John A Stankovic. “Impact of radio irregularity on wireless sensor networks”. In: *MobiSys’04*. ACM. 2004, pp. 125–138.

[133] Alexios Lekidis, Paraskevas Bourgos, Simplice Djoko-Djoko, Marius Bozga, and Saddek Bensalem. “Building distributed sensor network applications using BIP”. In: *Sensors Applications Symposium (SAS), 2015 IEEE*. IEEE. 2015, pp. 1–6.

[134] Gabriel Montenegro, Nandakishore Kushalnagar, Jonathan Hui, and David Culler. “Transmission of IPv6 packets over IEEE 802.15. 4 networks”. In: *RFC* 4944 (2007).

[135] Thomas Hérault, Richard Lassaigne, Frédéric Magniette, and Sylvain Peyronnet. “Approximate probabilistic model checking”. In: *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer. 2004, pp. 73–84.

[136] Luca Benini, Robin Hodgson, and Polly Siegel. “System-level power estimation and optimization”. In: *Proceedings of the 1998 international symposium on Low power electronics and design*. ACM. 1998.

[137] Ayoub Nouri, Balaji Raman, Marius Bozga, Axel Legay, and Saddek Bensalem. “Faster Statistical Model Checking by Means of Abstraction and Learning”. In: *RV’14*. Springer. 2014, pp. 340–355.

[138] Mathieu Jan, Christophe Jouvray, Fabrice Kordon, et al. “Flex-eWare: a flexible model driven solution for designing and implementing embedded distributed systems”. In: *Software: Practice and Experience* 42.12 (2012), pp. 1467–1494.

[139] Zhenyu Song, Mihai T Lazarescu, Riccardo Tomasi, Luciano Lavagno, and Maurizio A Spirito. “High-Level Internet of Things Applications Development Using Wireless Sensor Networks”. In: *IoT*. Springer, 2014, pp. 75–109.

[140] Alessandro Testa, Antonio Coronato, Marcello Cinque, and Juan Carlos Augusto. “Static verification of wireless sensor networks with formal methods”. In: *Eighth International Conference on Signal Image Technology and Internet Based Systems (SITIS)*. IEEE. 2012, pp. 587–594.

[141] Siyuan Xu, Weikai Miao, Thomas Kunz, Tongquan Wei, and Mingsong Chen. “Quantitative Analysis of Variation-Aware Internet of Things Designs Using Statistical Model Checking”. In: *Software Quality, Reliability and Security (QRS), 2016 IEEE International Conference on*. IEEE. 2016, pp. 274–285.

[142] Catello Di Martino, Marcello Cinque, and Domenico Cotroneo. “Automated generation of performance and dependability models for the assessment of wireless sensor networks”. In: *IEEE Transactions on Computers* 61.6 (2012), pp. 870–884.

[143] Sylvain Cherrier, Ismail Salhi, Yacine M. Ghamri-Doudane, Stéphane Lohier, and Philippe Valembois. “BeC3: Behaviour Crowd Centric Composition for IoT Applications”. In: *Mob. Netw. Appl.* 19.1 (Feb. 2014), pp. 18–32. ISSN: 1383-469X.

[144] Clément Duhart, Pierre Sauvage, and Cyrille Bertelle. “A Resource Oriented Framework for Service Choreography over Wireless Sensor and Actor Networks”. In: *International Journal of Wireless Information Networks* 23.3 (2016), pp. 173–186.

[145] Benjamin Bertran, Julien Bruneau, Damien Cassou, Nicolas Loriant, Emilie Balland, and Charles Consel. “DiaSuite: A tool suite to develop Sense/Compute/Control applications”. In: *Science of Computer Programming* 79 (2014), pp. 39–51.

[146] Pankesh Patel and Damien Cassou. “Enabling high-level application development for the internet of things”. In: *Journal of Systems and Software* 103 (2015), pp. 62–84.

[147] Nils Glombitza, Dennis Pfisterer, and Stefan Fischer. “Using state machines for a model driven development of web service-based sensor network applications”. In: *ICSE’10*. ACM. 2010, pp. 2–7.

[148] Amirhosein Taherkordi, Frédéric Loiret, Azadeh Abdolrazaghi, Romain Rouvoy, Quan Le-Trung, and Frank Eliassen. “Programming sensor networks using REMORA component model”. In: *International Conference on Distributed Computing in Sensor Systems*. Springer. 2010, pp. 45–62.

[149] Christos Antonopoulos, Katerina Asimogloy, Sarah Chiti, Luca D’Onofrio, Simone Gianfranceschi, Danping He, Antonio Iodice, Stavros Koubias, Christos Koulamas, Luciano Lavagno, et al. “Integrated Toolset for WSN Application Planning, Development, Commissioning and Maintenance: The WSN-DPCM ARTEMIS-JU Project”. In: *Sensors* 16 (2016).

[150] Ryo Shimizu, Kenji Tei, Yoshiaki Fukazawa, and Shinichi Honiden. “Model driven development for rapid prototyping and optimization of wireless sensor network applications”. In: *Proceedings of the 2nd Workshop on Software Engineering for Sensor Network Applications*. ACM. 2011, pp. 31–36.

[151] Haruhiko Kaiya and Motoshi Saeki. “Using domain ontology as domain knowledge for requirements elicitation”. In: *Requirements Engineering, 14th IEEE International Conference*. IEEE. 2006, pp. 189–198.

[152] Najah Ben Said, Takoua Abdellatif, Saddek Bensalem, and Marius Bozga. "A Model-Based Approach to Secure Multiparty Distributed Systems". In: *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I*. 2016, pp. 893–908.

[153] Miguel Andres Navarro Patino. "Energy efficiency in data collection wireless sensor networks". PhD thesis. Purdue University, 2016.

[154] Ionela Halcu, Grigore Stamatescu, and Valentin Sgârciu. "Enabling security on 6LoWPAN/IPv6 Wireless Sensor Networks". In: *7th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*. IEEE. 2015.

[155] Stylianos Basagiannis, Panagiotis Katsaros, and Andrew Pombortsis. "An intruder model with message inspection for model checking security protocols". In: *computers & security* 29.1 (2010), pp. 16–34.

[156] Tushar Deshpande, Panagiotis Katsaros, Stylianos Basagiannis, and Scott A Smolka. "Formal analysis of the DNS bandwidth amplification attack and its countermeasures using probabilistic model checking". In: *High-Assurance Systems Engineering (HASE), 2011 IEEE 13th International Symposium on*. IEEE. 2011, pp. 360–367.