Using Simulation to Validate Style-Specific Architectural Refactoring Patterns

Zoë Stephenson, John McDermid and Jason Choy
High-Integrity Systems Engineering Group
Department of Computer Science
University of York
Heslington, York YO10 5DD, UK
Telephone: +44 1904 432749
Email: zoe@cs.york.ac.uk

Abstract

When developing a new domain-specific architectural style, there can be uncertainty about the feasibility of using that style. In particular, the HADES architectural style contains refactoring patterns intended to remove undesirable scheduling features such as deadlock and livelock, but these patterns have not yet been validated. We report on the development of a simulator environment to help validate these refactoring patterns and generally demonstrate HADES architectures to non-specialists. The simulator implements the synchronisation and coordination specified by the architecture to help visualise the behaviour of the otherwise static architectural descriptions. We found simulation to be a useful tool in both visualising complex interaction semantics and in validating refactoring patterns.

1. Introduction

Software architecture is principally concerned with the organisation of software components and their interactions. Architectures are typically structured so that key emergent properties such as throughput and user response can be assessed. Our research focuses on the use of software in dependable embedded systems; examples of the properties of interest here are freedom from exceptions, data freshness, liveness, data accuracy and process schedulability. Software that is used in this fashion encompasses control and monitoring functions, hence its architecture is most often presented in a data-flow style.

A data-flow style expresses the architecture in terms of activity components, store components and data flowing between those components. Activity components perform transformations on the data, and each possible flow of data through the components is an individual transaction.

In a survey of architectural styles [18], Shaw uses an embedded cruise control system to compare the features of eleven architectural styles, all suited to the description of embedded control systems. Many of these styles use some form of data-flow diagram to show how the information flows through the control system. The Simulink tool [12] is a commonly-used simulation tool that specifies control systems in terms of transformation blocks and signals. The on-screen simulation models are similar to the control diagrams used in the control engineering profession, further suggesting that a data-flow style is to be preferred in this domain.

Embedded control systems often require particular types of formal analysis to demonstrate their correctness. The state of the art in comprehensive analysis of dataflow style architectures is exemplified by the Real-Time Networks (RTN) formalism [17]. Based on an earlier successful architectural effort, MASCOT [19], this style provides an underlying logic to define the sequences of interaction for particular protocols and activities.

HADES is a data-flow based architectural style originally intended for the representation and organisation of reusable SPARK Ada components [20]. It was inspired by the MASCOT family of architectural styles, but uses a simple transition-constraint semantics to synchronise component behaviours. Its original focus on reusability led to a flexible mechanism of implicit interface refinement, an area where systems engineering and software engineering approaches tend to exhibit large differences.

Systems engineering methods tend to treat all interfaces as fixed, potentially bidirectional connections, much like the pins on a panel connector or an integrated circuit package. At the systems level, interfaces are often prescribed by standards and customers, further reinforcing this static view. Interfaces are rarely altered; instead, unused connections are given new purposes, or whole new interfaces are added, sometimes crossing subsystem boundaries. Adding

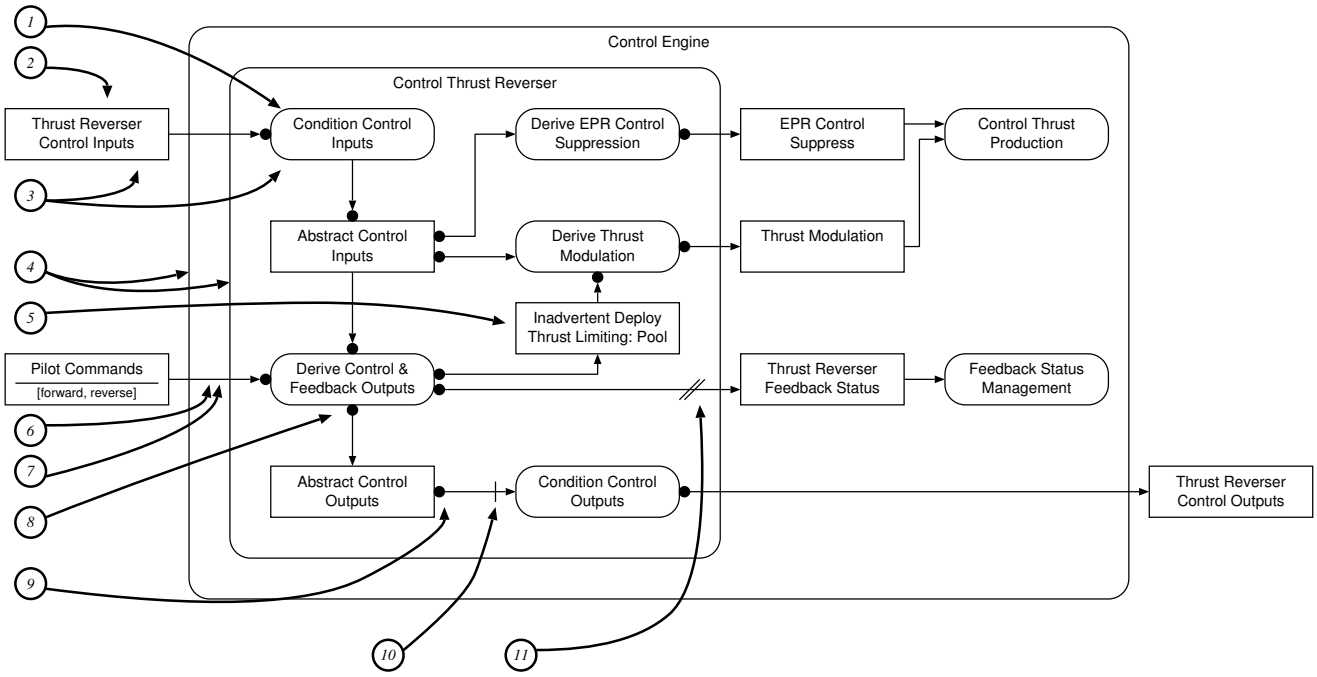

Figure 1. Example HADES diagram with labelled language features

new interfaces becomes especially important when dealing with safety-critical systems, as fault detection and accommodation often need access to information from multiple locations.

In the software world, interfaces are typically unidirectional; one package defines entry points (e.g. in the form of method or function calls) and any other package may make use of those entry points. Interfaces in object-oriented software systems are generally designed to be extensible; classes implementing the interface may provide specialisations and additional methods, for example. Encapsulation and information-hiding rules generally prevent information from crossing subsystem boundaries directly without some explicit forwarding mechanism.

Interfaces in safety-critical systems pose a problem for reuse. If the interface is in the systems engineering style, then reuse in a different context can be difficult — the failure characteristics of the whole system can change, leading to knock-on changes to the information needed for fault accommodation. Similarly, using a software engineering style of interface makes it difficult to route the information required for fault detection and accommodation through the architecture, as this generally means making data visible to modules for the sole purpose of passing on to other modules.

HADES attempts to provide an interface mechanism somewhere between the two, to provide for reuse in a safety-critical context. The data passed between components may be refined hierarchically, in a manner similar to adding new fields to a record structure; it may also be refined by adding new data that is not part of an existing interaction. The synchronisation and decomposition rules guarantee that commitments made in the architecture at one level are not broken by refinements of that architecture.

The diagram in Fig. 1 shows a typical HADES model, with numbered features as follows:

1. Activities are components represented using rounded rectangles.
2. Data areas are components represented using ordinary rectangles.
3. Activities communicate only with data areas, and data areas communicate only with activities.
4. Activities and data areas are graphically decomposed — data areas into further data areas, and activities into networks of activities and data areas.
5. Data areas can carry different protocols. The default data protocol is “Value”, a simple transmission of a value from one activity to another. Other options are “Pool” (shared data) and “Queue”.
6. Activities and data areas communicate over connectors, represented using directed arrows.
7. Data flows along a connector in the direction of the arrow.

8. The component responsible for initiating the communication has a filled circle on its end of the connector.

9. A data area may only initiate a communication in response to receiving data or a request for data from an activity.

10. Perpendicular lines on a connector are synchronous termination — the connection is held up until the data has been used.

11. Diagonal lines on a connector are asynchronous termination — the connection is terminated on delivery of the data.

HADES follows the tradition established in dataflow styles and the MASCOT-derived ADLs that data stores — even for data transmitted between just two components — are components in their own right. HADES connectors specify the direction of data flow and synchronisation of data access, and a structure that might be considered a connector in a different style may well be expressed in HADES as a connector, a data area component and another connector.

The underlying behaviour of activity and data area components is controlled by a set of transition-constrained finite-state automata, as in the example in Fig. 2. The diagram shows two activities communicating asynchronously over a passive data area. Below the diagram, each component is expanded into state machines with rectangles representing states and labelled vertical arrows representing transitions. Each label contains a number of single-letter identifiers and then a remark to document the label. The start state is replicated at the bottom so that the transition constraints are not cluttered by the transitions that return to the start state. The dotted lines on the diagram represent the flow of information, with a circle in the middle of the pool machine to represent the item being held in the pool.

Connectors are translated into constraints on transitions with particular labels, represented by lines perpendicular to those transitions. So, C1 is represented by constraints C↔Q, O↔I and D↔S; C2 by N↔Q, O↔I and D↔S. Each constraint prevents the connected transitions from being available unless all of those transitions are taken together. Further constraints can also be derived from particular arrangements of parent and child components to ensure that child components do not renege on the commitments of their parent component interfaces. The intent of these rules and semantics is to ensure that any arrangement of architectural elements will permit the flow of data without deadlock or livelock, and that potential processing bottlenecks may be identified for further analysis. However, it is trivial to construct an architecture in the HADES style that causes a deadlock; one such architecture is shown in Fig. 3. To overcome this issue without complicating the underlying semantics, a set of refactoring patterns is added to the style, as shown at the top of Fig. 4. Each of the models on the left exhibits a deadlock; the models on the right should perform the function intended of the models on the left, but without the associated deadlock.

For example, examine the left-hand model fpush. The intent is to perform a chain of calculations A1, A2, A3 and feed back the result from A3 as an input to the next iteration of A2. However, the components A2, D2, A3 and D3 all form a request loop, causing a deadlock. The preferred solution with a loop of this kind is to explicitly store the feedback parameter D3 in a data pool. A2 can read the value independently of A3 setting the value. The presence of a data pool should also prompt the designer to verify the first-pass behaviour of A2, a need that was not so readily apparent in the original structure.

These patterns are intended to eliminate common problematic structures while preserving the overall synchronisation in the architecture. This paper is concerned with the validation of these refactoring patterns. A pattern of this type is considered to be valid if:

- It identifies a problematic HADES structure that exhibits liveness, deadlock or some other synchronisation problem;
- Its solution does not exhibit any of these problems;
- Its solution exhibits the behaviour that was intended in the original structure. That is, apart from the changes to synchronisation, activities are able to process data from the same sources and produce the same kinds of outputs.

Three related technologies are useful for the determination of the above characteristics: theorem-proving, model-checking and simulation. While theorem-proving and model-checking are powerful techniques for the identification of the presence or absence of synchronisation problems, simulation offers an advantage in bringing out the dynamic behaviour of the architectural problem and solution being assessed. This is an important feature in helping to deliberate over the *intended* behaviour, and so the simulation approach is preferred for the validation of the HADES refactoring patterns. A simulator is a valuable tool for a number of other reasons:

- The on-screen animation of the architecture can provide a more accessible representation of the behaviour for system engineers and programmers.
- The use of simulation at the level of software is similar to the use of simulation at the system level to validate control algorithms; skills and mental models of behaviour are likely to be more easily learned and transferred.

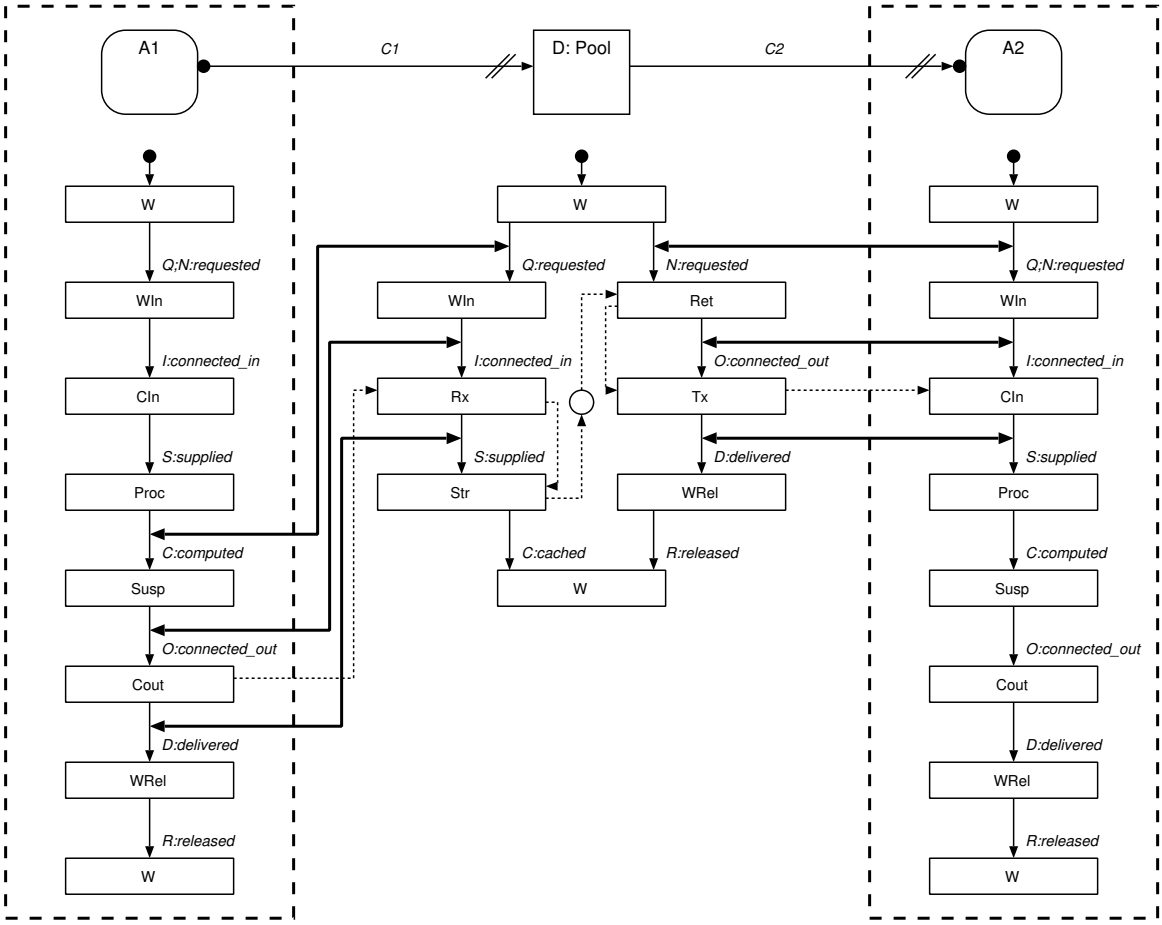

Figure 2. Example transition constraints in HADES

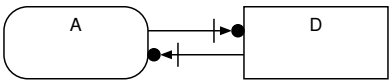

Figure 3. Example HADES diagram with deadlock; requests block waiting for data that relies on the result of those requests

- The simulator can show the state of each component, giving an explicit view of the underlying behaviour of the architecture; this helps in demonstrating the underlying semantics.

The following section reports briefly on the development of the simulator environment. Section 3 shows the use of the simulator to evaluate the refactoring patterns and to evaluate complex designs to which those patterns had been applied. In Section 4 we report on similarities with other work in the field, and our conclusions and ideas for further work are covered in Section 5.

2. Simulator Development

This section describes the development of the simulator environment. The simulator is intended only for the animation of HADES architectures, with support for connection to arbitrary scripting and analysis tools.

2.1. Test Cases

The test cases for the simulator environment came from three sources:

- The HADES report [20] contains four refactoring patterns to eliminate common structures that cannot be correctly synchronised. The pre- and post-refactoring structures in these patterns form the basis of the test suite.
- When discussing the underlying semantics of HADES with other researchers, there were two examples for which we could not agree on the intended behaviour. These curious diversions were added to the test suite.
- Two case studies were drawn from experience in dependable embedded systems design:
  - Missile launch control (adapted from a real-time network description [16]) — see Fig. 5.
  - Engine thrust reverse (using the model from the HADES manual [20, p35]) — see Fig. 1.

These case studies are used as a way of gaining confidence in the ability of the refactoring patterns to remove problematic architectural structures. Fig. 4 summarises the individual test models used.

2.2. Implementation

The simulator was developed in an agile fashion using Java. Consultation between the developer and the customer occurred at least once per day to ensure that issues were promptly addressed and requirements were clarified and expanded as needed. Java was chosen to help ensure portability and maintainability of the implementation.

The diagram in Fig. 6 gives an overview of the simulator implementation. Table 1 shows the different requirements areas and the design choices and patterns used to meet those requirements.

3. Evaluation

We presented 12 test cases in total to the simulator for evaluation. In each case, the simulator was used to determine the following properties:

- Did the simulation proceed so that components were able to move into and out of computation and storage states? Some arrangements of components were expected to "lock up" and not allow any part of the architecture to proceed.
- When activities used data, was that data from a consistent snapshot of the overall inputs?
- When activities were decoupled with pools and queues, did both activities get an equal chance to proceed with their computations?

For the test cases that represent the pre- and post-refactoring stages of the refactoring patterns, the pattern validation steps outlined in Section 1 were also applied, using the simulator output to determine whether the refactored architectural structure was consistent with the intent of the original structure. All evaluation properties were deduced manually from the individual simulation steps produced by the simulator.

The results of the evaluation are presented in Table 2. The columns of the table list the model, whether a deadlock was encountered, whether a deadlock was expected (all of the pre-refactoring models were expected to deadlock, for example), whether it was possible to determine the age of the data used in each activity, and whether the simulation was live (i.e. all of the components were able to cycle through their states).

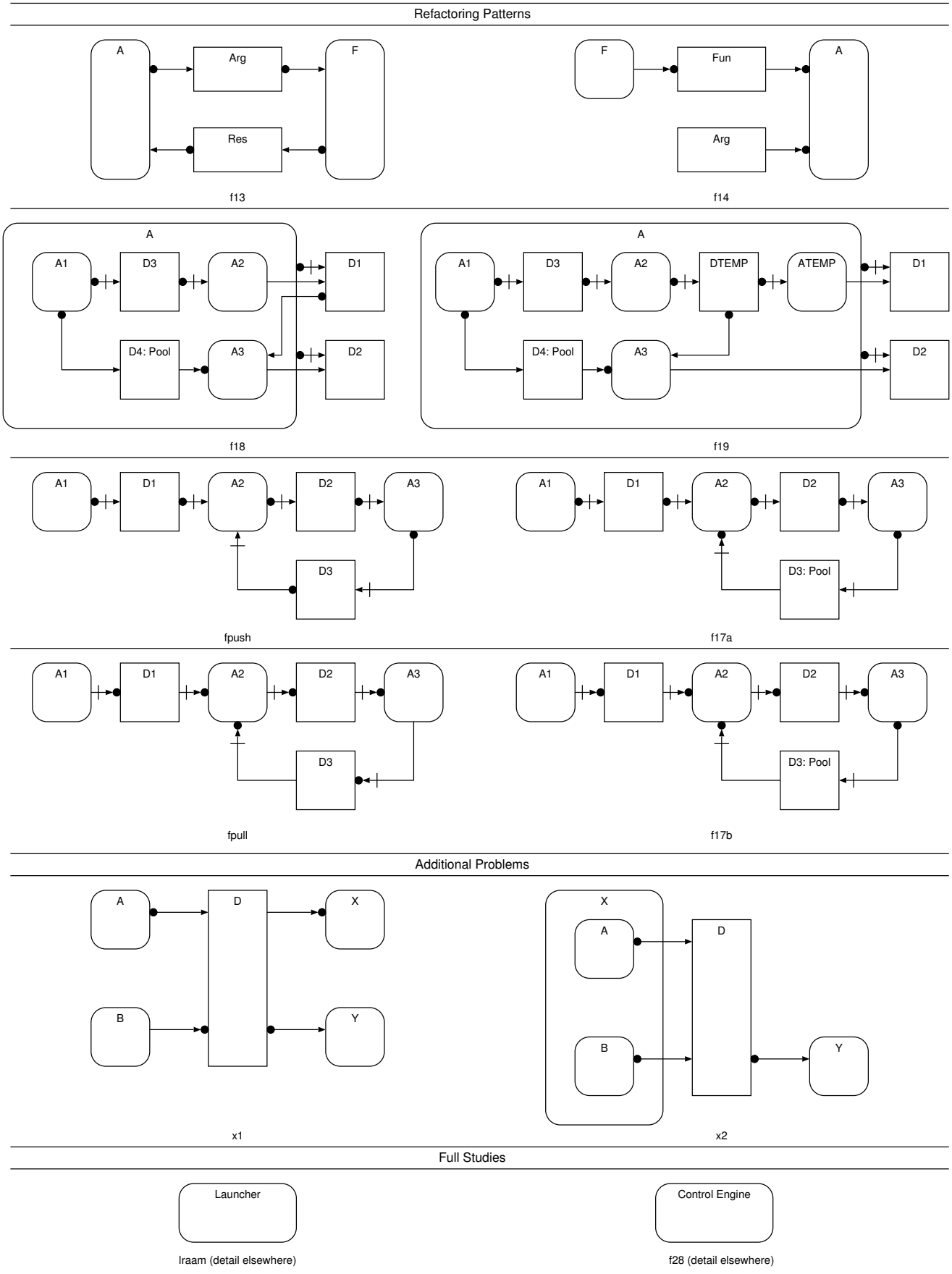

Figure 4. HADES models used to validate HADES semantics and refactoring patterns

Table 1. Requirements and design choices

Requirement	Design
Create a framework to access the existing XML representation of the HADES diagram and represent its objects.	The XML representation is accessed through a dedicated input/output system. The simulation passes an empty model to the input system when reading the model from the filesystem; the XML system uses the interface to the model to construct the components, connectors and constraints.
Analyse the diagram to determine validity and discover parent/child connector relationships.	Only syntactically legal diagrams may be constructed; the model facade rejects operations that would create an illegal diagram. Whenever a simulation is started, constraints are computed if the diagram has changed since the previous simulation.
Create a visualisation system to present the diagram to the user.	The user interface is separated from the model using the observer pattern. The user interface controls all aspects of the visual presentation
Create a model of activity behaviours such as normal-distribution outputs or continuous transmission.	Activity behaviour is created using a specific behaviour component attached to the activity. The use of a separate component allows for the simulation of architectures with different behaviours — for example, components can be tied to a global clock.
Create user interaction to store and retrieve settings such as behaviour definitions for activities.	The simulation facade presents a method that causes the output of simulation details such as behaviour definitions to a file.
Run simulations until terminating conditions occur, until no data can be produced, or for a specified time, at the user's control.	The simulation steps the state of each part of the model according to the transition-constrained automata semantics. The user interface can query the state of the simulation through that interface and terminate as needed.
Store simulation results in a log file.	The simulation facade presents a method that causes the output of simulation results to a given log file.
Display component states and data tokens on the main diagram during simulation.	The simulation facade allows the user interface to observe the state of each component and its data content. Component observers show the current state alongside the name; connector observers show the movement of data tokens by animating a visual representation along the path of the connector. Tokens carry a representation of their path through the architecture to assist with analysis.
Display the full state machine and the transition constraints for each component and connector during simulation.	The definition of the state machine layout for each component type is held in a separate class, for use by the user interface when rendering the layout during simulation. The separation ensures that definitions are shared when appropriate, and allows for flexibility in the way state machines are shown. The same data could also be used to export the model as a complete state machine for other tools.
Allow cue/review/record/retrieve in the user interface to view simulation results.	The simulation stores a snapshot of the model at each simulation step. The interface to the simulation allows the user interface to retrieve a specific simulation step for rendering.
Allow full editing of the diagram in the user interface.	The user interface is part of an overall model-view-controller pattern that allows for editing of the diagram using the simulation tool.

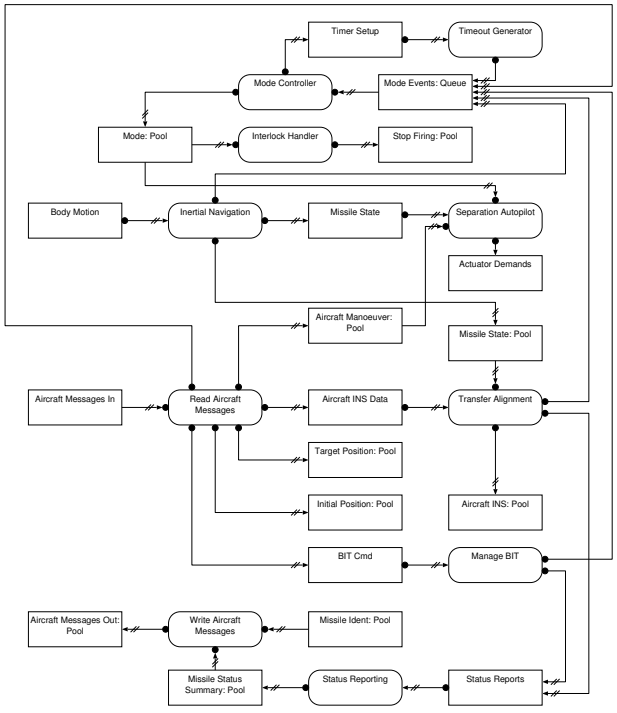

Figure 5. An LRAAM Launcher Design (translation from Figure 3 of the LRAAM documentation [16])

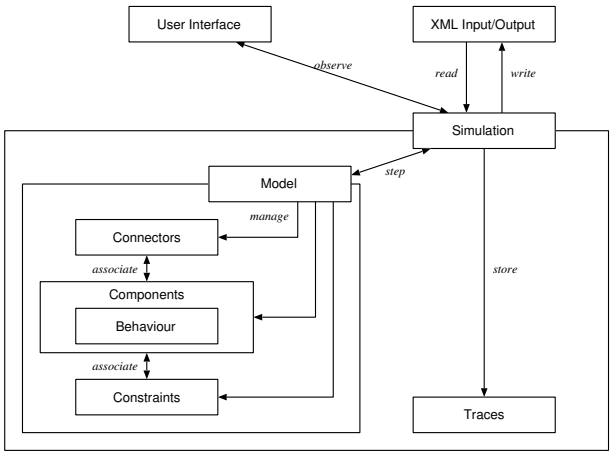

Figure 6. Simulator overview

Table 2. Simulation results

Model	Deadlock	Expected	Freshness	Liveness
f13	Yes	Yes	-	-
f14	No	No	Yes	Yes
f18	Yes	Yes	-	-
f19	No	No	Yes	Yes
fpush	Yes	Yes	-	-
f17a	Yes	No	-	-
fpull	Yes	Yes	-	-
f17b	Yes	No	-	-
x1	No	Yes	Yes	Yes
x2	No	No	Yes	Yes
lraam	No	No	Yes	No
f28	Yes	No	-	-

The evaluation of the refactoring patterns is shown in Table 3. Here, the columns show the pair of models taken from the pattern, whether the deadlock was detected, whether the refactoring eliminated the deadlock and whether the intent of the model was preserved. The entries labelled "Yes*" indicate that while the deadlock was not eliminated by the pattern as described, the only changes were to synchronisations; all of the components remained the same, and the directions of their data transfers remained the same.

The simulator was used to diagnose the synchronisation problems with the f17 models; the cause was identified as the loop of synchronous terminations on the connectors. Weakening one of the connections to asynchronous termination allowed these models to operate without deadlock. Other problems that were observed during simulation included:

- The original version of the model shown in Fig. 1 did not use the pool type for the data area "Inadvertent Deploy Thrust Limiting". This led to a deadlock as the request from "Derive Thrust Modulation" and the supply from "Derive Control & Feedback Outputs" were out of step. Changing the data area to a pool decoupled the interaction properly.

- In that same diagram, the parent activities "Control Engine" and "Control Thrust Reverser" share no inputs or outputs with their child activities. This allows those parent activities to clock through their states without regard for the behaviour of their subcomponents. This phenomenon does not interfere with the ability to simulate the model, but is unintuitive.

- In the f17 models, the data paths along which data tokens are collected include a cycle. Each time round that cycle, the tokens accumulate the full history of previous cycles. The implementation had anticipated this in part by storing references to the paths of existing tokens; however, this does still place a structure- and memory-dependent limit on the number of steps for which a simulation may be run. The simulator would benefit from a comprehensive treatment of architectural cycles.

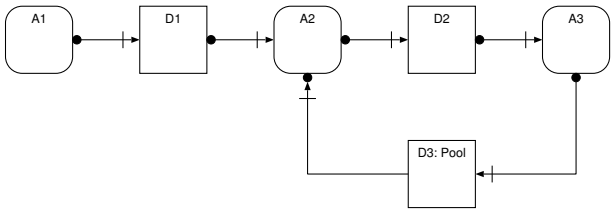

Figure 7. Feedback loop pattern

In the majority of these cases, the simulator provided crucial insight into the existence and nature of a problem resulting from the combination of the state-based semantics and arrangements that appear, at the informal level, to be valid. The modifications listed above were adequate in correcting these misunderstandings and allowing all of the test cases to run.

The validation of the four refactoring patterns showed that the transformations identified in those patterns did not all result in working HADES models. However, for those refactoring patterns that did not work (the feedback loop patterns with unnecessary synchronous terminations) a simple alternative was found that resulted in the desired outcome. Hence, while the original description of the pattern was incorrect, the simulator was able to help in both identifying the incorrectness and validating that the corrected version worked as expected.

One further anomaly that was not expected was the presence of livelock in the missile launcher model. This was due to the use of data pools at the edges of the model, with one side constrained by model activity and the other side completely unconstrained. This allowed the pools to continually cycle around the unconstrained path rather than engaging with any of the activities. Several ideas were generated for ways of correcting this situation, including virtual activities, additional behaviour descriptions to impose fairness criteria or new data area types to represent external inputs and outputs. There was not time during the simulator project to try any of these suggestions, however.

The original goal of validating the refactoring patterns through simulation has been achieved. In addition, corrections were made to these patterns and to the original case study model that was used to guide the hand-coding of the original SPARK Ada implementation. The simulator has also been useful in demonstrating the concepts of the HADES architectural style to colleagues.

Table 3. Pattern evaluation results

Refactoring Pair	Deadlock Identified	Deadlock Eliminated	Intent Preserved
f13, f14	Yes	Yes	Yes
f18. f19	Yes	Yes	Yes
fpush, f17a	Yes	No	Yes*
fpull, f17b	Yes	No	Yes*

4. Related Work

In the field of software architecture, the domain of dependable embedded systems is well-represented. The MASCOT [19] style was one of the first styles specific to this domain, and is the inspiration for much of the structure of HADES. Activities, explicit data areas and the various data area protocols are all drawn from MASCOT. The Real-Time Network formalism [17] expanded on the MASCOT ideas by providing an underlying logic with which to describe its semantics. The logic is more general than the state-machine description used in HADES, but the connection between the architecture and the results of the analysis is perhaps less obvious to the casual user.

HRT-HOOD [5] is an object-based style with hierarchical decomposition, synchronous and asynchronous messages and explicit methods. It fits well with Ada-based implementations, and provides some inspiration for HADES in its hierarchical decomposition and message-passing concepts.

Most architectural styles in use within the research community are defined using some kind of formal semantics, and that set of semantics is typically only concerned with interactions. Taking Acme [6] as one example:

> ...Acme relies on an open semantic framework that provides a basic structural semantics while allowing specific ADLs to associate computational or run-time behaviour with architectures...

The semantic framework for Acme is based on relations and constraints. There are other architectural description languages that were developed in the same era; for example, Wright [2] uses an event/process notation based on CSP [8]. The focus of this architectural style is on completeness and consistency of interfaces. Event-based definitions are also found in Rapide [10], where the semantics are defined by partially-ordered sets of events. Here, the focus is on animation and event traces in order to detect anomalous behaviours. The Darwin style [11] is based on the π-calculus and is geared towards hierarchical distributed processing. In each case, the designer of the language has chosen a formal basis for that language that reflects the properties of interest in a particular domain.

The C2SADEL [14] approach is more mature, based on experience in evolving architectures for rapid development using off-the-shelf components. Its formal basis has been kept to a minimum in favour of practical concerns with the description of large and complex architectures. It uses a type theory and first-order predicate logic to describe the structure and behaviour (precondition, postcondition and invariant) of the architecture.

AADL [3] is an emerging standard for avionics architecture systems, derived from the MetaH [21] language. The main formal constructs of the MetaH language are types and state machines; the types define properties that are used during analysis. AADL is particularly concerned with schedulability and fault-tolerance, and uses a model of the underlying hardware platform during analysis.

Simulation is widely used for the analysis of physical computer hardware, from individual processors to distributed networks. However, it rarely appears as a way of dealing with software architecture. Indeed, in a comprehensive survey of software architecture styles [13], simulation was not included in the classification scheme. The concept does appear in Bosch's work on software architecture and product lines [4], but only in terms of implementing the architecture in a representative simulation environment before deployment.

Model-checking for architectural descriptions is generally targeted at properties such as liveness and mutual exclusion [7]. The formal basis of Giannakopoulou's work on the model-checking of Darwin architectures is similar to HADES in its use of finite-state automata, and that basis is also similar to RTN in its use of linear temporal logic. The CBabel language [9] is aimed at the verification of coordination, distribution and quality of service in concurrent architectures, and uses precondition/postcondition semantics. It also uses reflection to impose architectural constraints on an implementation, in a manner similar to an aspect-oriented language. Model-checking is also seen in static correctness and completeness checks for the Wright language [1].

5. Conclusions and Further Work

HADES is an architectural style for dependable embedded systems offering the software architect an implicit-interface style of decomposition while decoupling transaction control through the architecture. It was found to be useful in the development of a sample thrust reverser sys-

tem [20] but required further validation before it would gain acceptance for wider use in the dependable systems domain. The work reported in this paper describes a simulation environment targeted at demonstrating the validity of the HADES refactoring patterns. We believe that this simulator environment is a valuable tool in understanding both the complex operational semantics of HADES and the refactoring patterns for HADES architectures.

We intend to carry out further validation of the HADES ideas by using a model-checking environment to assess worst-case scenarios for liveness and data freshness properties. This should provide a basis for the integration of HADES with other projects within our research group.

The hierarchical nature of HADES and its use in software for dependable embedded systems implies that HADES will eventually be integrated with standard system-level environments such as Simulink. Although the decomposition paradigm is slightly different, we remain optimistic that HADES can provide for the analysis of important implementation characteristics for systems created using Simulink. An alternative strategy may be to leverage the SysML standard [15] as a way of unifying software-level and system-level analyses.

References

[1] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, School of Computer Science, Jan. 1997.

[2] R. Allen and G. Garlan. Formalizing Architectural Connection. In *Proceedings of the Sixteenth International Conference on Software Engineering*, pages 71–80, May 1994.

[3] R. Allen, S. Vestal, D. Cornhill, and B. Lewis. Using an Architecture Description Language for Quantitative Analysis of Real-Time Systems. In *Proceedings of the Third International Workshop on Software and Performance*, pages 203–210, July 2002.

[4] J. Bosch. *Design and Use of Software Architectures*. Addison-Wesley, 2000.

[5] A. Burns and A. Wellings. *Hard Real-Time HOOD: A Structured Design Method for Hard Real-Time Ada Systems.* Elsevier, 1995.

[6] D. Garlan, R. Monroe, and D. Wile. Acme: An Architecture Description Interchange Language. In *Proceedings of CASCON*, 1997.

[7] D. Giannakopoulou. *Model Checking for Concurrent Software Architectures*. PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, University of London, 1999.

[8] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[9] O. Loques, A. Sztajnberg, J. Leite, and M. Lobosco. On the Integration of Meta-level Programming and Configuration Programming. *Lecture Notes in Computer Science*, (1826):191–210, June 2000.

[10] D. C. Luckham, J. J. Kenney, L. M. Augustin, et al. Specification and Analysis of System Architectures Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, Apr. 1995.

[11] J. Magee and J. Kramer. Dynamic Structure in Software Architectures. In *Proceedings of ACM SIGSOFT'96: Fourth Symposium on the Foundations of SoftwareEngineering*, pages 3–14, Oct. 1996.

[12] T. MathWorks. Simulink — Simulation and Model-Based Design. `http://www.mathworks.com/products/simulink`.

[13] M. Medvidovic and R. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, Jan. 2000.

[14] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A Language and Environment for Architecture-Based Software Development and Evolution. In *Proceedings of the International Conference on Software Engineering*, 1999.

[15] S. Partners. SysML Specification. `http://www.sysml.org/artifacts.htm`, Jan. 2005.

[16] S. E. Paynter. A BVRAAM Case Study in Safety Engineering: Specification and Design. Technical Report 23121, MBDA Missile Systems, Dec. 2001.

[17] S. E. Paynter, J. A. Armstrong, and J. Haveman. ADL: An Activity Description Language for Real-Time Networks. *Formal Aspects of Computing*, 12(2):120–140, Feb. 2000.

[18] M. Shaw. Comparing Architectural Design Styles. *IEEE Software*, 12(6):27–41, Nov. 1995.

[19] H. R. Simpson. The MASCOT Method. *Software Engineering Journal*, 1(3):103–120, 1986.

[20] Z. R. Stephenson and D. L. Buttle. The HADES Architectural Style — Development and Definition. Technical Report YCS 373, University of York Department of Computer Science, Feb. 2004.

[21] S. Vestal. MetaH User's Manual. Honeywell Technology Center, 1998.