Practical Issues with Formal Specifications
Lessons Learned from an Industrial Case Study

Michael Altenhofen and Achim D. Brucker

SAP Research, Vincenz-Priessnitz-Str. 1, 76131 Karlsruhe, Germany
{michael.altenhofen,achim.brucker}@sap.com

Abstract. Many software companies still seem to be reluctant to use formal specifications in their development processes. Nevertheless, the trend towards implementing critical business applications in distributed environments makes such applications an attractive target for formal methods. Additionally, the rising complexity also increases the willingness of the development teams to apply formal techniques.
In this paper, we report on our experiences in formally specifying several core components of one of our commercially available products. While writing the formal specification, we experienced several issues that had a noticeable consequences on our work. While most of these issues can be attributed to the specific method and tools we have used, we do consider some of the problems as more general, impeding the practical application of formal methods, especially by non-experts, in large scale industrial development.
Keywords: ASM, industrial case study, formal specification.

1 Introduction

In this paper, we report on experiences we made with writing a formal specification for certain aspects of an application that had been designed and built by one of our product groups. Given the actual time and resource constraints, we did not attempt to write a full-fledged specification that would allow us to (semi-)automatically prove system properties, but rather opted for an *executable* specification that would help us gaining further insights into the behavior of the system via proper simulation runs. This both seemed feasible and desirable, especially since the target application has to operate in a cluster environment where testing and debugging is notoriously difficult.

Based on the experiences we had made in previous research [4], we decided to use abstract state machines (ASMs) [7] for our formalization. In more detail, we created a set of specifications where the refined version could eventually be executed in CoreASM [13]. Since CoreASM comes with built-in support for *literate specifications* (similar to literate programming [15]), we wrote a document that contained extensive documentation explaining the specification. The final version of that document accumulated to roughly 130 pages containing approximately 3 200 lines of CoreASM specification (code).

During the course of writing this specification we stumbled across several issues that had a noticeable influence on our work in general and the resulting specification in particular. While most of them can be clearly attributed to the method and tools we used, some of them seem to show more general problems that cannot be avoided by simply changing the underlying formal method. In this sense, we believe that our case study outlines several challenges that need to be tackled to foster the application of formal software specification methods in industrial product development environments.

The rest of the paper is organized as follows: In Sec. 2, we start with a short description of the application that we want to specify followed by an outline of the approach we have taken to eventually arrive at an executable formal specification. We then address the issues we have witnessed during the course of writing the specification in Sec. 3 and, finally, we conclude in Sec. 4.

2 Case Study: Distributed Object Management

In this section, we give an abstract description of the component we have specified formally. This component is part of an enterprise application that is built by one of our product development teams. Moreover, we briefly summarize the requirements that constrained the developers while designing and implementing the application.

2.1 The Problem: Consistent Distributed Object Management

In its essence, the application under consideration implements an event-condition-action *rule engine* [17], where events are represented as object state changes, conditions are formulated as expressions on object attributes, and actions lead to further changes in object states. To efficiently compute the actions that need to be executed on events, the engine uses a modified version of the Rete algorithm [14] that propagates object state *deltas* through Rete networks. The actual implementation is multi-threaded, so access and updates to objects need to be coordinated among a potentially large set of threads running concurrently.

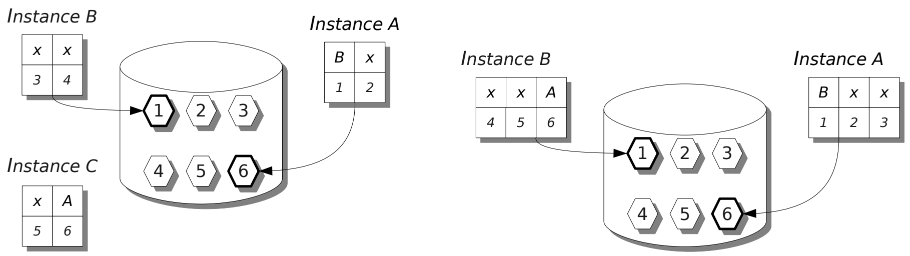
If the engine runs in a non-distributed setting, i. e., a single application instance, optimistic locking provides exclusive read/write access to the different objects. The engine, however, may be deployed in a cluster variant, where multiple application instances are running on different *server nodes*. In this case, we need to consistently coordinate object access across these engine instances.

Although the overall cluster size may be fixed, the system exhibits dynamic behavior in that application instances may start or stop during the overall lifetime of the cluster. Thus, we need mechanisms to deal with variations in the cluster topology, especially in the case of unexpected changes due to application or cluster node failures.

2.2 The Solution: Object Ownership and Cluster Failover Management

The implemented system does not use a distributed locking protocol, but rather tries to coordinate object access among different instances by maintaining meta information, called *object ownership*, in a shared data structure.

The solution needs to guarantee *exclusive* object ownership, i. e., at most one application instance may work with an object at any point in time. Thus, any application instance that wants to access or modify an object needs to successfully acquire ownership for that object from its current owner. As scalability is an important property of the overall system, the data structure that keeps track of ownership information is not maintained by a central instance, but managed in a distributed manner: Each application instance is responsible for managing ownership information for a fixed subset of all objects and is called *authoritative indexer*[1] for this set of objects. Fig. 1a illustrates a scenario with three application instances A, B, and C: instance A is authoritative indexer for objects 1 and 2, B for objects 3 and 4, and C for objects 5 and 6. Objects do not need to be owned by their authoritative indexer. In our example, object 1 is owned by instance B, object 6 by instance A, while all other objects are unused.



(a) Instance A is authoritative indexer for object 1 and 2, instance B for 3 and 4, and instance C for 5 and 6. Object 1 is owned by instance B, object 6 by instance A, while all other objects are unused.

(b) After instance C has left, instance A and B agree on a cluster of size 2, with A being authoritative indexer for objects 1 to 3, and instance B for objects 4 to 6. Instance B is informed that object 6 is owned by instance A.

Fig. 1. An example of a cluster distributed over several application instances

If an instance wants to acquire ownership for an object, it does so by always contacting the authoritative indexer of that object, not the current owner (if there is any). This approach has two advantages: first, the protocol requires at most two message exchanges; one from the requesting instance to the authoritative indexer and one from the authoritative indexer to the current owner. And

[1] Indexer refers to the fact that objects have unique identifiers that serve as an index into this data structure.

second, each cluster instance is able to compute all authoritative indexer by itself once it has learned the cluster topology after a successful join of the cluster.

If cluster topology changes, ownership and authoritative indexer information needs to be redistributed among all cluster members. Ownership information is propagated via a restructuring protocol that, upon successful completion, is supposed to ensure two (mutually independent) properties:

1. all participating instances will agree on the same view (i. e., size and topology) of the cluster, which allows each instance to locally compute *the same* authoritative indexer for any object.
2. each instance will know current ownership for its authoritative set of objects.

To illustrate this, recall our example in Fig. 1a. If instance C leaves the cluster, instance A and B will eventually agree on a new cluster of size 2, with A being authoritative indexer for objects 1, 2, and 3, and instance B for objects 4, 5, and 6. Furthermore, instance B has to be informed that object 6 is owned by instance A. So far, that information had been maintained by the leaving instance C. Fig. 1b illustrates the resulting cluster topology.

While each instance maintains its local view of the cluster, there is one dedicated *master* instance providing the current view of the cluster to new instances joining the cluster. Using a dedicated master avoids (cluster) discovery protocols, but requires explicit means for *master election*, including recovery mechanisms in case the current master instance may leave the cluster unexpectedly due to an application or server node failure. In those cases, the remaining instances will compete against each other regarding mastership and the failing parties will try to join the cluster in the usual way.

2.3 Additional Implementation Constraints

Application development, especially in large software companies, rarely happens in isolation. Overall, it has to obey various requirements and boundary conditions imposed by application frameworks and platforms and programming models that are already used. These constraints often have a noticeable effect on the resulting solution architecture. In this section, we briefly review those aspects that also had a significant impact on our formal specification work.

Avoid additional functionality by reusing existing frameworks. Rather than building dedicated functionality into the runtime environment, the development team was urged to implement functionality by reuse existing software frameworks and components as much as possible. While it is, e. g., desirable to have a central facility for storing cluster meta data, like information on the current cluster topology or on the current master instance, much like [11], the given cluster implementation does not foresee such mechanisms. Therefore, the team opted for *named* communication channels implemented using the Java Naming and Directory Interface (JNDI).

Minimize Central Knowledge while Avoiding Redundancy. Centralized knowledge requires additional synchronization among the cluster participants and increases the communication overhead among them. Furthermore, any form of centralized knowledge introduces bottlenecks and threatens system availability should the central instance stop working properly. A common practice to increase system availability is redundancy (e. g., via replication [12]), but such a feature is not part of the underlying runtime platform. Therefore the decision was made to solely rely on local meta information (i. e., object ownership) per instance which needs to be synchronized whenever the cluster topology changes (which is expected to happen rarely).

Global Synchronization via Locks. Whenever an operation requires synchronization among the instances in a cluster, the initiating party needs to enforce that by acquiring a global lock maintained by a central lock server (i. e., a central infrastructure component). Master election is an example for such an operation. In fact, there is no real election going on and no elaborated agreement protocol is used; instead, being able to become the master is just bound to the ability to acquire a global, exclusive *master lock* from the central lock server.

Synchronous Mode of Operation. Although the platform provides different means of communication for application instances running on a cluster, any protocol-related communication is implemented as synchronous remote method invocation (RMI) calls since that required less changes in the code base when moving from a stand-alone to a cluster-enabled version.

Continuous Operation during Restructuring. Obviously, the restructuring protocol for updating meta information on object ownership is one of the most critical parts of the overall solution. A defensive approach would probably try to block any other interfering operations (like object requests) during cluster restructuring until the system has reached a stable state again. But overall performance had been given higher priority leading to a significantly more complicated restructuring protocol.

2.4 Formal Specification

Ideally, we would have started with that formal specification, proven its correctness, and then iteratively refined it into executable code. Unfortunately, the real project settings were different and the development team had already designed and implemented a first version. Given that, we opted for a rather practical approach: our goal was to reverse-engineer the implementation into an executable specification that would allow us to simulate the system behavior in enough detail to detect any discrepancies between the desired and the implemented behavior. The initial plan was to focus on robustness of the protocol against communication failures. During our work, we followed a two-staged approach:

Table 1. An overview of the modules of the ASM specification

Module	Lines	Rules	Functions
Control ASM States	50	0	1
Cluster Master	161	12	10
Protocol Messages	138	0	25
Cluster Membership and Object Management	1 796	114	159
Object Requests	128	10	3
Cluster Environment (Notification)	328	19	31
Lock Management	141	7	19
Message Passing	362	12	56
Control Flow	88	10	5
Control State Handling	63	5	9
Total	3 255	189	318

1. We started with a high-level abstract specification on paper to capture the essence of the functional features. This abstract specification was used as our primary communication and discussion medium with the development team to clarify our understanding of the overall system architecture and behavior and to discuss remaining open issues.
2. Once that abstract specification had reached a critical mass, we began to manually refine it towards an executable specification. After that, we updated both versions in parallel while trying to keep the overall structure and naming conventions aligned. Although this does by no means replace any sort of formal proof of the correctness of our refinement, it eventually helped us correcting errors in the abstract specification that surfaced through simulation runs of the executable specification.

Given the dynamic nature of the application, we decided to model the system as an *asynchronous multi-agent ASM*. With this, we came fairly close to the implementation where the parallelism induced by multiple Java threads was mapped to a set of agents with dedicated functionality. As a positive side-effect, this also led to a more modular specification.

Within a time-period of six months, we spent 80 person days to write a multi-agent CoreASM specification that eventually consists of ten modules. Tab. 1 provides some details on the complexity of those modules. Out of these ten modules, the first four resemble the basic functionality outlined in Sec. 2.2. The fifth module, Object Requests, has been added to trigger random object access requests and thus simulate updates on ownership information. The next three modules provide functionality available via application frameworks (see Sec. 2.3), while the last two have been introduced to provide "syntactical sugar" when it comes to specifying complex control state machines. As the numbers show, we ended up with roughly 20% additional effort not providing core functionality, but is required to realistically model the implemented system behavior.

Table 2. The different agents per node and their number of control states

Agent	Control States
Object Requester	8
Object Request Processor	5
Node Failure Handling	22
Meta Data Management	15
Joining a Cluster	22
Leaving a Cluster	24

For each cluster node, we have six agents performing different tasks in the overall protocol and each agent is modeled as a control state ASM (see Tab. 2). Since some of the control states are shared between these agents, the overall number of distinct states is 79.

Specification 1.1 presents invariants (in CoreASM notation) which must hold whenever a cluster is considered in a stable state, i. e., no nodes are in the process of joining or leaving the cluster: As the names imply, we want to assert that, at any point in time, object ownership information is "in sync" and "valid" across the cluster. Synchronized information requires that, for each object, its authoritative indexer and its current owner share that view. Ownership information is considered valid if the current owner is still a member of the current cluster.

```
derived IndicesInSync =
  forall node in RunningNodes() holds IndexInSync(node)

derived IndicesAreValid =
  forall node in RunningNodes() holds IndexIsValid(node)

derived IndexInSync(node) =
  forall oid in [1..OID_MAX] holds SlotInSync(node, oid)

derived SlotInSync(node, oid) =
  node = authIndexer(oid, node) implies
    OWNER(OWNER(node, oid), oid) = OWNER(node, oid)

derived IndexIsValid(node) =
  forall oid in [1..OID_MAX] holds SlotIsValid(node, oid)

derived SlotIsValid(node, oid) =
  node = authIndexer(oid, node) implies
    OWNER(node, oid) memberof RunningNodes()
```

Specification 1.1. Cluster Protocol Invariants

2.5 Simulation Results

Given the specification above, a rough estimate shows that the state space required by a explicit state model checker is the range of 10^{50}. Thus, we rather went for simulating dedicated scenarios, instead explicit brute-force model checking.

As with any other distributed coordination protocol, it soon became clear that we needed to simulate protocol runs for exceptional cases, especially situations where nodes leave the cluster unintentionally. When we started our work, we thought we would need to spend most of our efforts into simulating message transmission errors. But after several talks with the development team it turned out that the system takes a fairly defensive approach for dealing with such errors: most of the time, a message transmission failure will lead to a node restart. Thus, we decided to focus on exploring the alternative paths with regard to cluster topology changes and failover handling.

As it turned out, the modularity of the specification came in very handy and we were able to factor out parts of the overall protocol complexity, like, e. g., object request handling. With this simplifications, we eventually ended up with a streamlined simulation scenario that revealed a bug in the initial implementation, not yet discovered by any standard testing procedures: While investigating the failover handling during changes of the cluster topology, we realized that the original failover protocol was based upon a faulty assumption, namely that notifications in the case of failure would be sent *immediately after a node failure*. As this notification is sent by the runtime environment and, thus, not controlled by the application, one can easily think of scenarios where this is not true. Just assume that the notification is delayed while a new node is starting up in parallel during that delay. Then, that node will become the master of a new cluster that would just consist of that one node. If the delayed notification is then passed on to the remaining nodes from the old cluster, they will try to become master, will all fail, and thus do nothing, assuming that the (unknown) winner will perform the outstanding restructuring. Since the new master is not aware of the old cluster, no repair will happen and we will end up with two independent clusters operating in parallel.

This undesired behavior can be reliably reproduced with the following abbreviated simulation scenario.[2] We start by setting up a cluster with two nodes. Furthermore, we specify a distinct id for a third node that will be started at a later stage and will become the new master of the new cluster.

```
if (scenarioPhase = 0) then {
  nodeList = ["N1", "N2"]
  newMasterNode = "N3"
        scenarioPhase = 1
}
```

[2] We have omitted some variable and rule declarations. The overall simulation script is 89 lines long.

Once these nodes are running, we know that the cluster has reached a stable state. We now disable node failure detection, by suspending the corresponding agents.

```
if (scenarioPhase = 1) then {
  if (AllNodesRunning()) then {
      SuspendNodeFailureHandlers()
      scenarioPhase := 2
      clusterIsStable := true
  }
}
```

After failure detection has been disabled, we forcefully shutdown the current master node.

```
if (scenarioPhase = 2) then {
  killedMaster := MasterNode()
  remove NodeID(MasterNode()) from nodeList
  SignalNodeShutdown(MasterNode(), true)
  scenarioPhase := 3
  clusterIsStable := false
}
```

As soon as the old master node is down, we start up the third node. Failure detection is still disabled, i. e., the remaining node in the old cluster is still not informed about the fact that the old master has left the cluster.

```
if (scenarioPhase = 3) then {
  if (NodeIsDown(killedMaster)) then {
    AddNode()
    add newMasterID to nodeList
    scenarioPhase := 4
  }
}
```

Once the new master has joined the cluster and there is a (new) master in that cluster, we resume the agents that will handle node failures.

```
if (scenarioPhase = 4) then {
  if (MasterNode() != undef
      and HasJoinedCluster(newMasterID)) then {
    ResumeElemLossHandlers()
    scenarioPhase := 5
  }
}
```

As a result of the previous step, the one remaining node of the old cluster will try to become master, but will fail (since the new node has taken over mastership). Assuming that another node from the old cluster has become master, it will do nothing. Once the remaining and the new master node have resumed normal operation, we declare the cluster as stable again. But now the invariant `IndicesInSync` does not hold anymore[3].

```
  if (scenarioPhase = 5) then {
     if (AllNodesRunning()) then {
        clusterIsStable := true
        scenarioPhase := 6
    }
  }
```

3 Lessons Learned

Although we have ultimately reached our goal, it turned out to be more difficult than we expected. Some of the issues we have faced can clearly be attributed to the method we have used, while other seem to reveal more general problems.

3.1 Method-Related Issues

Notation and Execution Semantics. As noted above, all protocol-related communication is implemented as synchronous RMI calls, which means that the calling thread will *block* until the answer has been received from the callee. Translating this blocking behavior into ASM turned out to be difficult. At the abstract level, we finally ended up with extending the standard semantics by introducing an `await` construct (see [2] for details) and, moreover, provided additional control state diagrams for further explanation.

Alas, this approach could not be taken for the executable specification since that would have required substantial changes in the existing CoreASM runtime. Instead, we transformed the corresponding rules and state diagrams from the abstract specification into proper control state ASMs. To increase readability, we ultimately developed a set of ASM macros that allowed us to use a more concise notation, as the following example shows:

```
rule JoinCluster = {
  StepInto(@PrepareJoin, {startingUp, registerAtMaster})
  StepInto(@Rearrange, {arrangeCall})
  StepInto(@Commit, {rearrangeCompleted})
}
```

Here, the `StepInto` macro has the following semantics: If the control state of the agent is a member of the state set specified in the second argument, the *program* of that agent shall be overridden by the rule element specified in

[3] In the implementation, an authoritative indexer claims ownership for all unassigned objects within its range. In our scenario, the new master—as the sole member of the new cluster—will claim membership for all objects, which conflicts with ownership information maintained by the old node.

the first argument. In other words, the first line in the example states that the agent should "step into" `PrepareJoin` if its control state is either `startingUp` or `registerAtMaster`.

Missing Scope for Locations. Although the ASM method provides a detailed classification scheme for functions and locations [7], we missed a way to restrict the scope or visibility of a location to an individual agent or a well-defined subset of agents. This feature would have allowed us to make constraints that exist in the implementation already visible (and checkable) at the specification level.

In Sec. 2.2, we outlined that the system requires "shared" information to operate correctly, but that implies that each application instance maintains its local copy of that information and any changes need to be propagated among the instances via proper message exchanges. Without having a way to attribute information as being "private" to an instance (similar to private fields in object-oriented languages), one could easily introduce errors in the specification by accidentally accessing such private information in other contexts.

Missing Tool Support for Refinements. As the name suggests, abstract specifications should provide a high-level view capturing the essential functionality of a system. We took the freedom to "abstract away" implementation related issues during the initial phase of our work. Compared to that, the executable specification had to spell out all the details that we had left out in the abstract specification. That constitutes a large refinement and should have probably been broken up into several steps. Unfortunately, none of the tool sets that were available to us does provide any support for controlled refinements.

Faced with that problem, we again took a rather pragmatic approach: we tried to establish a strong linkage on the syntactical level by staying as close as possible to the naming conventions and signatures introduced in the abstract specification although we could have used a more concise notation in some cases. There are, e. g., abstract rules which are parametrized with a *node* referring to the application server node which will process a request. In the executable specification, we do have functions that establish a unique relationship between an agent and such a server node. Given that relationship, the node parameter in the CoreASM rule signature is redundant, but has been retained to keep the rule signatures synchronized.

Reusable Specification Modules. When writing our specification, we often encountered situations in which we needed to specify common concepts (e. g., asynchronous communication channels) that, with respect to our target, we would classify as "infrastructure." Based on our experiences with programming languages that are equipped with large, thoroughly tested libraries of common data structures and algorithms, we often felt the need for similar libraries of well-proven, generic specifications of common software engineering artifacts. Consequently, we tried to make our specification as re-usable as possible; still, we cannot claim that our specification can be easily reused in other contexts than our own. We believe that is partially due to our own lack of experience in writing modular ASM specification and partially due to the lack of generic modules in ASM. Finally, while systems like Isabelle [18] or Coq [6] provide a large variety of re-usable libraries formalizing mathematical concepts, we still see a lack of similar libraries for data-structures, algorithms, and high-level components (e. g., of-the-shelf middleware components). Besides being the basis for further formalization work, such libraries of standard components and algorithms could also serve as means for learning how to write good specifications. Thus, we would especially encourage initiatives collecting and maintaining formal specifications for software artifacts, similar to "The Archive of Formal Proofs" (`http://afp.sourceforge.net`) for Isabelle.

3.2 Tool- and Process-Related Issues

While the topics above can be attributed to the specific method we have chosen, we also see deficits when it comes to development tools and processes used and established in industrial environments.

Insufficient Support for Literate Specifications. Within our work, we have experimented with the literate specification feature in CoreASM: We embedded the executable specification into a document written in OpenOffice.org (`http://www.openoffice.org`) which should allow us to use the full power of a modern desktop publishing system to improve the comprehensiveness of the formal part with diagrams, cross-references, etc. The CoreASM runtime engine is able to extract the specification part from such a document and directly execute it.

While this loose coupling seems flexible and elegant at first sight, it has proven inferior in both usability and efficiency: On one hand, an editing environment that is unaware of the specification language syntax lacks many of the sophisticated features, like syntax highlighting, auto-completion, etc., found in modern, integrated development environments, such as Eclipse (`http://www.eclipse.org`). On the other hand, having no real feedback loop between the editing front-end and the runtime back-end unnecessarily prolongs the round trip for error corrections in comparison to state-of-the-art development tools. In hindsight, we would prefer a tight integration into existing tool environments over such loosely coupled tool chains.

Although there are first examples of tools that strive for better integration into existing environments, e. g., the Rodin platform (`http://www.event-b.org/platform.html`) for Event-B [1], support for literate specifications still seems to be lacking behind. We still see a tendency to follow the tradition to treat a formal specification as part of an (academic) *publication*. In Rodin, e. g., there is no easy way to export a machine specification other than exporting it to LaTeX (via a separate plug-in). But for large-scale application development, we need a way to make a formal specification a *living document* within the overall development life-cycle.

Debugging Support. When writing specifications one often has to cope with situations similar to programming. Like programs, specifications may have bugs, and finding these bugs may require a deeper inspection of what is going on. While simulation support primarily asks for ways to steer execution runs and have a way to observe the externally visible state changes, debugging support would extend this towards the possibility to fully explore the state of the specification execution.[4] Such a fine-grained specification animation helps, on the one hand, in convincing oneself (and, in our case, also the developers) that the formal specification captures the informal requirements and, on the other hand, it allows for finding the inconsistencies ("bugs") in the specification in an early stage.

For example, we envision support for executing deterministically specified traces within the animation environment while being able to set breakpoints for examining the system state (e. g., variables, messages sent). As a first step in that direction, our experiences in simulating ASM runs in CoreASM resulted in the development of a scripting language for CoreASM that is discussed elsewhere [3]. Overall, this scripting language allows for deterministically provoking the bug described in Sec. 2.5 by performing the following steps automatically:

1. Start a cluster with two nodes and wait until it has reached a stable state.
2. Disable node failure notification.
3. Stop the master node.
4. Once the master node is done, start a new node.
5. Once that new node has finished building the new cluster, enable node failure notification.

In our experience, such "scripted" traces are also very helpful in communicating with the developer of the analyzed product.

Combining Formal and Semi-formal Development Processes. Whereas formal methods are far from being deeply integrated into our software development process, semi-formal methods, e. g., in the form of UML or BPMN are used routinely. Therefore, these already existing, semi-formal specifications should be reused in a tool-supported way. This could be done either by providing formal methods tool for these languages and integrating them into model-driven development processes (e. g., similar to [9,8]) or by generating specifications in the formal language of choice. Such generated specifications could describe, on the one hand, the environment, and on the other could serve as the basis for a formal high-level system specification.

Lack of Commercially Applicable Tools. While being a completely non-technical issue, we experience amazingly often the situation in which the software license of a tool prevented its use—even for case-studies. Either, while being available for download, the tools did not have any licensing information (which, at the end, forbids their use) or because the use in a commercial environment is excluded explicitly in the license terms (and, furthermore, no option for obtaining a commercial license is provided). Thus we would like to encourage tool

[4] Lacking that feature in CoreASM, we fell back to the "traditional" way of debugging by augmenting the specification with logging statements. In the final version, roughly 10% of the whole specification are dedicated to produce meaningful execution traces.

developers to state their intended license terms clearly. In our experience, this is especially important to advertise the use of formal methods in environments that are not able (either due to a lack of resources or expertise) to develop their own tools. For example, today's (rare) use of formal methods at SAP is too diverse to suggest a concrete formal toolchain (and specification language) to our product groups. Thus, we would like to use formal tools from external vendors, similar to our uses of development tools (e. g., for Java development) from external vendors. Consequently, we see a higher chance to educate our product groups in using SAT or SMT solvers[5] for specific problems than writing formal specification of whole software components.

4 Conclusion

Fully automated tools, that apply formal methods without the need for an explicit specification (neither of the underlying software system or of the properties to be analyzed), e. g., Polyspace (`http://www.mathworks.com/products/polyspace/`) or Coverty (`http://www.coverty.com/`), can be used by non-experts in formal methods [19]. Similar experiences are reported for automated tools that only require light-weight specifications (e. g., based on pre-, postconditions and invariants) on the level of source code annotations that enjoy a deep integration into the development life-cycle, e. g., [5].

In our experience, the use of formal specifications, within an industrial software development process for business software, using languages like ASM [7], B [1], or Z [20], is still a challenge. While we do not see a fundamental problem in requiring an expert for the (potential) interactive analysis (e. g., verifying system properties), non-experts should be able to document, write, type-check, and animate (execute) formal specifications and the system properties that should be verified during an analysis. Overall, to achieve this goal, the specification and animation environment needs to be integrated into modern software development tool chains used in industry. Moreover, as software is usually developed in, potentially distributed teams, support for a collaborative writing of specifications seems to be a necessity. This is in particular true if existing software development teams work closely together with formal methods experts.

Overall, we see in particular four areas for future research: First, the integration of collaboration techniques, e. g., wikis[6], into environments for writing specifications would allow for turning formal specifications from nicely formatted (academic) papers into living documents. While there are first experiments in integrating interactive theorem provers into a semantic wiki for generating formally checked pages [16], we still see this only as a first step. Collaborative

[5] At SAP, using SAT solvers, at least for prototypes, seems to be an accepted development approach. Nevertheless, due to technological and licensing issues it is still unclear if a solution based on a SAT solver will make its way into shipped products or if, during production, they might be replaced by a customized analysis algorithm.

[6] There is another interesting aspect to this: wikis have successfully proven that a simplified notation can significantly extend the user base.

scenarios with distributed teams (of developers and formal method experts) may require sophisticated life-cycle and versioning support that would allow teams to develop, refine and test several specification variants in parallel.

Second, software changes over time and the same should be true for its accompanying documentation and formal specification. Therefore, a tool-supported process that (automatically) ensures consistency and traceability among all dependent artifacts is, in our opinion, a central cornerstone of the successful application of formal specifications in the mainstream software industry.

Third, we would like to stress once again the importance of a library mechanism allowing for both the structuring of specifications and, more importantly, the reuse of already analyzed specifications. Similar to the component libraries available for programming language, such libraries need to be easily available within the regular tool chain (e. g., similar to the handling of Java libraries in Eclipse), reusable, covering a wide application area (ranging from data structures, over algorithms and protocols, to high-level specifications of large components, e. g., middleware), and, last but not least, available to the public.

Finally, we see a potential for integrating test case generation techniques (e. g., similar to [10]) into specification and animation environments. This would allow for both the generation of test cases on the level of the specification and the generation of test cases on the specification level. While the former allow for validating that the implementation–including the environment it is executed in—is a refinement of the specification, the latter can be used for guiding the animation of the specification.

Acknowledgments. We would like to thank Egon Börger, who helped in writing the abstract specification, and Roozbeh Farahbod, who provided substantial support to us in successfully using CoreASM.

The work described here was partly supported by the EU research project ICT 214158 DEPLOY (Industrial deployment of system engineering methods providing high dependability and productivity)[7]. The authors are responsible for the content of this publication.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Design. Cambridge University Press, New York, NY, USA (2009)
2. Altenhofen, M., Börger, E.: Concurrent abstract state machines and ^{+}CAL programs. In: Recent Trends in Algebraic Development Techniques, Lecture Notes in Computer Science, pp. 1–17. Springer-Verlag, Heidelberg (2009). doi: 10.1007/978-3-642-03429-9_1
3. Altenhofen, M., Farahbod, R.: Bârun: A scripting language for coreasm. In: M. Frappier, U. Glässer, S. Khurshid, R. Laleau, S. Reeves (eds.) ASM, *Lecture Notes in Computer Science*, vol. 5977, pp. 47–60. Springer-Verlag, Heidelberg (2010). doi: 10.1007/978-3-642-11811-1_5

[7] http://www.deploy-project.eu/

4. Altenhofen, M., Friesen, A., Lemcke, J.: ASMs in service oriented architectures. Journal of Universal Computer Science **14**(12), 2034–2058 (2008)
5. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: G. Barthe, L. Burdy, M. Huisman, J.L. Lanet, T. Muntean (eds.) Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS, pp. 49–69. doi: 10.1007/b105030
6. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Springer-Verlag, Heidelberg (2004)
7. Börger, E., Stärk, R.F.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer-Verlag, Heidelberg (2003)
8. Brucker, A.D., Doser, J., Wolff, B.: An mda framework supporting ocl. Electronic Communications of the EASST **5** (2006)
9. Brucker, A.D., Wolff, B.: HOL-OCL – A Formal Proof Environment for UML/OCL. In: J. Fiadeiro, P. Inverardi (eds.) Fundamental Approaches to Software Engineering (FASE08), no. 4961 in Lecture Notes in Computer Science, pp. 97–100. Springer-Verlag (2008). doi: 10.1007/978-3-540-78743-3_8
10. Brucker, A.D., Wolff, B.: HOL-TestGen: an interactive test-case generation framework. In: M. Chechik, M. Wirsing (eds.) Fundamental Approaches to Software Engineering (FASE09), no. 5503 in Lecture Notes in Computer Science, pp. 417–420. Springer-Verlag (2009). doi: 10.1007/978-3-642-00593-0_28
11. Burrows, M.: The chubby lock service for loosely-coupled distributed systems. In: OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation, pp. 335–350. USENIX Association, Berkeley, CA, USA (2006)
12. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. ACM SIGOPS Operating Systems Review **41**(6), 205–220 (2007). doi: 10.1145/1323293.1294281
13. Farahbod, R., Gervasi, V., Glässer, U.: CoreASM: An extensible ASM execution engine. Fundamenta Informaticae **77**(1-2), 71–103 (2007)
14. Forgy, C.L.: Rete: A fast algorithm for the many patterns/many objects match problem. Artificial Intelligence **19**(1), 17–37 (1982). doi: 10.1016/0004-3702(82)90020-0
15. Knuth, D.E.: Literate programming. The Computer Journal **27**(2), 97–111 (1984). doi: 10.1093/comjnl/27.2.97
16. Lange, C., McLaughlin, S., Rabe, F.: Flyspeck in a semantic Wiki. In: C. Lange, S. Schaffert, H. Skaf-Molli, M. Völkel (eds.) SemWiki, *CEUR Workshop Proceedings*, vol. 360. CEUR-WS.org (2008)
17. McCarthy, D., Dayal, U.: The architecture of an active database management system. In: SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data, pp. 215–224. ACM Press, New York, NY, USA (1989). doi: 10.1145/67544.66946
18. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL—A Proof Assistant for Higher-Order Logic, *Lecture Notes in Computer Science*, vol. 2283. Springer-Verlag, Heidelberg (2002). doi: 10.1007/3-540-45949-9
19. Venet, A.: A practical approach to formal software verification by static analysis. Ada Lett. **XXVIII**(1), 92–95 (2008). doi: 10.1145/1387830.1387836
20. Woodcock, J., Davies, J.: Using Z: Specification, Refinement, and Proof. Prentice Hall International Series in Computer Science. Prentice Hall (1996)